Write Up

Comments

Code

Operating Systems Homework #4

Gabriel Groover, gtg3vv@virginia.edu

I was able to successfully implement a simple UNIX shell in accordance with the specifications.

1. Problem

The goal for this assignment was to build a UNIX shell that could parse and execute simple commands. The shell should be able to handle invalid input, multiple commands piped together, and I/O redirection.

2. Approach

The solution was developed in a single C++ file with an accompanying header file. This was done to take advantage of C++ standard library data structures such as strings and vectors. The design of the shell can be broken down into two parts: validation and execution.

Validation was done prior to execution of any commands. This allowed the execution portion of the code to make assumptions about the location of arguments and operators in each line. Shell.cpp includes a main function that repeatedly reads a line of input and verifies its length and contained characters before passing that input off to each of the following helper functions:

vector<string> tokenize(string s, const char* delim) – This is a customized split function that looks for each instance of delim in s, and pushes the portion of the string before the delimiter onto a vector.

string removeSpace(string s) – This function strips all leading and trailing spaces, as well as removing repeated spaces inside s. It returns a new string.

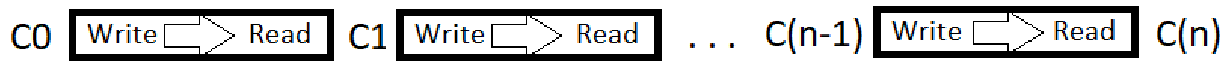
bool isValid(string inputLine) – This handles the bulk of the validation. Each input line is stripped of white space and then tokenized on " | ". From then on, each token in the group can be treated as a single command with its associated arguments and file redirects. The function then iterates through every token in each token group to ensure that group follows all the rules outlined in the specification. Input and output file operators are only allowed in the first and last token in the group respectively. There also may not be more than one of any operator, or an input operator after an output operator. All operators must be followed by a file, and there cannot be any arguments other than a file after the first operator. If every token group adheres to the rules and does not contain illegal characters, isValid will return true.

Once isValid has returned true, the remainder of the code assumes all input lines are of the form:

Command0 [arg0,arg1,…argn][< infile][> outfile] | Command2 … |

The lines are then split once again on " | " to create a list of *n* commands. The remainder of the work is handled by the main function.

For *n* commands piped together, *n-1* pipes are required. Each command *i* (except the last) requires a pipe between itself and command *i+1*. Command *i* must transfer its output from stdout to the write end of pipe *i*, and command *i+1* must transfer its input from stdin to the read end of pipe *i*. This allows the result of each command to be sent through the pipe to the next command as shown below.



The file descriptors for these pipes are kept in an int array of size *2*n*. The main function initializes each of the pipe descriptors such that the read end of pipe *i* is at index *2*i*, and the write end is at index *2*i+1*.

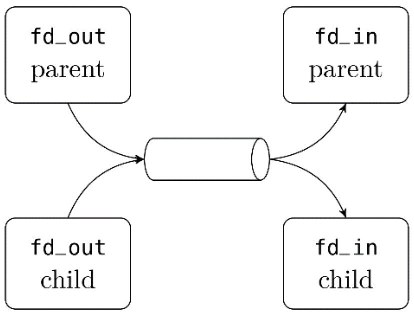
At this point, the main function iterates through each command in the input line and forks a child process for that command. The child process iterates through the tokens for the command and copies each token before the first operator into an argument vector. It also handles relative paths by appending the current working directory to the command if necessary. If any I/O operators were encountered, the filename that follows the operator is opened and the corresponding file descriptor is placed into stdin/out using dup2.

Finally, the child process connects any necessary pipes and executes the command. An arbitrary command *i*, will be reading its input from the read end of pipe *i-1*, and writing its output to the write end of pipe *i*. To ensure all processes can terminate, the child closes the ends of the pipes it is not using. Therefore, it closes the write end of pipe *i-1* and the read end of pipe *i*. The remaining read and write ends are copied to stdin and stdout respectively. After doing so, the command is executed with the argument vector constructed above. The parent process must then wait for each process to terminate and store the result codes in an array.

3. Problems Encountered

Most of the development process went smoothly. Each of the components were tested incrementally to catch small errors. However, two major problems occurred when implementing pipes. When initializing the array of pipes, a for loop had been used to iterate over the array by twos. The bounds of the loop though, stopped at n (where n is the number of commands in a line), causing only the first half of the array to be initialized. Input lines with a single pipe would terminate correctly, but any commands with more than one would hang forever. This was fixed by inspection of the code.

The other problem encountered was with closing pipes. When the child process is created for each command, it shares the same open file table as the parent. The parent has its own copy of each pipe file descriptor and must close them. If the parent process does not close the write end of each pipe it creates, child processes will hang expecting more output at the read end of the pipe.



Each of these problems were simple to fix but required a substantial amount of time due to poor assumptions about which part of the code was breaking.

4. Testing

As mentioned above, the code was developed incrementally. Each helper function was tested with a variety of string inputs to ensure it would return the correct values. The entire algorithm for validation was drawn out and refined several times before coding anything at all. Once the validation process seemed to be working, it was tested with all the example inputs from the specification.

The same whiteboarding process was done with the execution portion of the code. File redirection was done first and tested on single command inputs. Then piping was implemented and tested with multiple command inputs. Finally, all pieces were connected and tested again to ensure the resulting output and status code was correct.

5. Conclusion

This assignment emphasized the value of using pipes when coding. The process executing each command did not need to “know” where it was getting its input from or handle it any differently than normal. Each process is effectively just a module that can be placed anywhere in a chain.

It was also extremely beneficial to plan, develop and test cyclically. The implementation was concise, but complicated, and good coding practices helped make the process easier.

I pledge that I didn’t give or receive aid on this.

```
/*
 * Gabriel  Groover (gtg3vv)
 * HW4 - Simple Shell
 * Due: 3/17/2018
 * shell.cpp
 */

#include "shell.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>

using namespace std;

//Split string into vector based on delim
vector<string> tokenize(string s, const char* delim)
{
    vector<string> tokens;
    size_t start = 0;
    size_t loc = s.find(delim);

    //While delim in string
    while (loc != string::npos)
    {
        //Add string from last delim to current to vector
        tokens.push_back(s.substr(start, loc-start));
        start = loc+strlen(delim);

        loc = s.find(delim, loc+1);
    }
    //Push remainder
    tokens.push_back(s.substr(start, s.length()));
    return tokens;
}

//Remove leading/trailing/doubled spaces
string removeSpace(string s)
{
    string oneSpace = "";
    bool afterSpace = true;
    size_t lastLetter = s.find_last_not_of(" ");

    for (int i = 0; i < s.length(); i++)
    {
        //If repeated space, skip
        if (s[i] == ' ' && afterSpace)
           continue;

        afterSpace = s[i] == ' ';
        if (i <= lastLetter) oneSpace += s[i];
    }
    return oneSpace;
}

//Validate line
bool isValid(string inputLine)
{
    vector<string> groups = tokenize(inputLine, " | ");
    bool valid = true;

    //Iterate over token groups
    for (int i = 0; i < groups.size(); i++)
    {
        vector<string> tokens = tokenize(groups[i], " ");
        bool allow_operator = false; //Op allowed at position i
        bool allow_argument = true; //arg or command allowed at i
        bool foundIn = false; //found filein operator
        bool foundOut = false; //found fileout operator

        //Check leftover pipes
        if (groups[i].find_first_of("|") != string::npos)
            valid = false;


        //Iterate over tokens in group
        for (int j = 0; j < tokens.size(); j++)
        {
            if (tokens[j].find_first_of("<>") == string::npos)
            {
                if (allow_argument) allow_operator = true;
                else valid = false;

                allow_argument = !(foundIn || foundOut);
            }
            else
            {
                if (!allow_operator) //Operator at beginning or after other op
                    valid = false;
                else if (tokens[j].compare(">") != 0 && tokens[j].compare("<") != 0) //If extra chars around op
                    valid = false;
                else if (tokens[j].compare(">") == 0 && (!(i == groups.size()-1) || foundOut)) //If not last item
or already found
                    valid = false;
                else if (tokens[j].compare("<") == 0 && (!(i == 0) || foundOut || foundIn)) //If not first item
already found <>
                    valid = false;
                else
                {
                    allow_operator = false;
                    allow_argument = true;
            }

            foundIn = foundIn || (tokens[j].compare("<") == 0);
            foundOut = foundOut || (tokens[j].compare(">") == 0);
        }
    }
    //If never had file arg
    if (allow_argument && (foundIn || foundOut)) valid = false;
  }

  return valid;

}

//Main input loop
int main() {

  char temp[100];
  string cwd = string(getcwd(temp, 100));

  //Read input until eof
  while (true)
  {
    string inputLine;

    fflush(stdout);
    cout << ">";
    getline(cin, inputLine);

    if (inputLine.empty() || inputLine.compare("exit") == 0)
      return 0;

    inputLine = removeSpace(inputLine);

    //Validate input line
    if (inputLine.length() > 100)
    {
      cout << "ERROR: Input line too long" << endl;
      continue;
    }
    if
(inputLine.find_first_not_of("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz01
23456789-._/<>| ") != string::npos)
    {
      cout << "Error: bad char" << endl;
      continue;
    }
    if (!isValid(inputLine))
    {
      cout << "ERROR: invalid input" << endl;
      continue;
```

```
}

vector<string> groups = tokenize(inputLine, " | ");
vector<int> exitCodes;
vector<string> commands;
int pipes[(groups.size()-1)*2];
int pids[groups.size()];
int lastPid = -1;

//Init pipes
for (int i = 0; i < groups.size() - 1; i++)
{
   pipe(&pipes[2*i]);
}

//Iterate over token groups
for (int i = 0; i < groups.size(); i++)
{
   //Fork child for command
   pids[i] = fork();
   if (pids[i] == 0)
   {
      //Find last argument index
      vector<string> tokens = tokenize(groups[i], " ");
      int countArgs = 0;
      for (int j = 0; j < tokens.size(); j++)
      {
         if(tokens[j].find_first_of("<>") != string::npos)
            break;
         countArgs++;
      }

      string curCmd = tokens[0];
      //Add cwd if needed
      if (tokens[0][0] != '/')
         tokens[0] = cwd + "/" + tokens[0];

      //Build argv
      char **argv = new char*[countArgs+1];
      for (int j = 0; j < countArgs; j++)
      {
         char *temp = new char[tokens[j].length() + 1];
         strcpy(temp, tokens[j].c_str());
         argv[j] = temp;
      }
      argv[countArgs] = NULL;

      //Handle file redirects
      int infile, outfile;
      for (int j = countArgs; j < tokens.size(); j++)
      {
        if (tokens[j].compare("<") == 0)
        {
          infile = open(tokens[++j].c_str(), O_RDONLY | O_NONBLOCK);
          if (infile == -1)
          {
            fprintf(stderr, "Failed to open file for reading\n");
            exit(1);
          }
          dup2(infile, 0);
        }
        else if (tokens[j].compare(">") == 0)
        {
          outfile = open(tokens[++j].c_str(), O_RDWR | O_CREAT, 0666);
          if (outfile == -1)
          {
            fprintf(stderr, "Failed to open file for writing\n");
            exit(1);
          }
          dup2(outfile, 1);
        }
      }

      //Pipe to next process
      if (i == 0) //First
      {
        close(pipes[2*i]);
        dup2(pipes[2*i+1], 1);
      }
      else if (i != groups.size() - 1) //Middle
      {
        close(pipes[2*(i-1)+1]);
        close(pipes[2*i]);
        dup2(pipes[2*(i-1)], 0);
        dup2(pipes[2*i+1], 1);
      }
      else //Last
      {
        close(pipes[2*(i-1)+1]);
        dup2(pipes[2*(i-1)], 0);
      }

      //Handle exec or fail
      execv(tokens[0].c_str(), argv);
      fprintf(stderr,"Command %s failed to execute\n", curCmd.c_str());
      exit(1);
    } else
    {
      //Cleanup for parent
      commands.push_back(tokenize(groups[i]," ")[0]);
      if (i != 0)
      {
```

```
                    close(pipes[2*(i-1)]);
                    close(pipes[2*(i-1)+1]);
                }
            }
        }

        //Wait for each child
        for (int i = 0; i < commands.size(); i++)
        {

            int status;
            waitpid(pids[i], &status, 0);
            status = WEXITSTATUS(status);
            exitCodes.push_back(status);
        }

        //Print exit codes
        for (int i = 0; i < commands.size(); i++)
        {
           fprintf(stderr, "%s exited with exit code %d\n", commands[i].c_str(), exitCodes[i]);
        }
    }
}



/*
 * Gabriel  Groover (gtg3vv)
 * HW4 - Simple Shell
 * Due: 3/17/2018
 * shell.h
 */

#ifndef SHELL_H
#define SHELL_H

//Included libraries
#include <string.h>
#include <iostream>
#include <vector>

//Methods
std::vector<std::string> tokenize(std::string s, const char* delim); //Split a string into a vector based on
delim
std::string removeSpace(std::string s); //Trim whitespace from ends and remove double spaces
bool isValid(std::string inputLine); //Check if a given input line is a valid format

#endif
```