CS4414 (SP2017) Operating Systems

Project 1

Title: Writing Your Own Shell
Due: February 28, 2017 (12:30 p.m.)
Points: 20

(1) The Problem

Write a simple Linux shell in C/C++ that can (*i*) run programs and (*ii*) support commands for file redirection and pipes.

(2) Shell Specification

This section outlines the requirements for the shell that you are writing. You are allowed to work on the project in pairs; but groups of more than two are NOT allowed. You must generally work on the code together, with both collaborating on each segment of code (instead of working on different segments). You should ask questions for clarification if you think you have found an error or ambiguity (or even get lost) in the specification. Don't be intimidated by the length of the specification; it simply helps you write a simple (and working) version of a Linux shell syntax that you have always used. So let's get started!

(2.1) The Shell Language

(2.1.1) The Lexical Structure

- The input to the shell is a sequence of **lines**. The shell must correctly handle lines up to 80 characters. If a line containing more than 80 characters is submitted to the shell, it should print some kind of error message and then continue processing input at the start of the next line.
- Each line consists of **tokens**. Tokens are separated by one or more spaces. A line may contain as many tokens as can fit into 80 characters.
- There are two kinds of tokens: **operators** and **words**. The only **operators** are < (*i.e.* input redirection operator), > (*i.e.* output redirection operator), and | (*i.e.* pipe).
- **Words** consist of the characters A-Z, a-z, 0-9, dash, dot, and underscore. If a word in a line of input to the shell contains any character not in the set, then the shell should print an error message and then continue processing input at the start of the next line. Whenever you output an error (e.g. syntax error, parsing error, etc.), you must prefix your error message with "ERROR: " (without the quotes), and this message cannot exceed a single line.

- The only legal input to the shell, other than lines consisting of valid tokens, is when **exit** is entered at the prompt, which will terminate your shell program.

(2.1.2) Parsing the Shell Language

- Lines of input are divided into **token groups** separated by a **pipe** operator.
- Each token group will result in the shell forking a new process and then executing the process.
  e.g. cat –n myfile.txt | sort // two token groups separated by a pipe operator
- Every token group must begin with a word that is called the **command** (see example above). The words immediately following a command are call **arguments** and each argument belongs to the command it most closely follows (there are two arguments for the **cat** command above).
- It is permissible for the arguments in a token group to be followed by none, one (*i.e.* a "<" operator or a ">" operator), OR exactly two file redirections (*i.e.* a "<" operator followed by a ">" operator). A file redirection consists of one of the operators "<" or ">" followed by a single word called the **file**. A file redirection containing the "<" operator is an **input** file direction and a file redirection containing the ">" operator is an **output** file redirection.
- Token groups are separated by **pipe** operators. In other words, each valid line of shell input must begin with a valid token group, and the only place pipe operators are allowed is in between token groups.
- Valid token groups may be preceded by a pipe operator (e.g. ls | sort), or they may contain an input file redirection (e.g. sort < infile), or they may contain an output redirection (e.g. ls > outfile), or they may contain an input file redirection followed by an output file redirection (e.g. sort < infile | wc > outfile), or an input file redirection followed by any number of pipes (e.g. ls –l < infile | sort | uniq), or an input file redirection followed by any number of pipes followed by an output file redirection (e.g. sort < infile | grep hi | uniq > outfile). You may assume that the token following the redirection operators is always a file token (*i.e.* not a command token).
- Lines of shell input that violate any of the parsing rules above should cause the shell to print an error message and then move on to the next line of input.

(2.1.3) Some Examples of Shell Commands

- **ls -l**: a legal line of input. It contains a single token group containing two tokens.
- **ls -l|**: an illegal line of input because the pipe character is not valid as part of a word (remember that tokens are always separated by one or more spaces).
- **ls -l |**: an illegal line of input because the pipe operator has to separate valid token groups. It is not legal for a pipe operator to be at the end of a line of input.

- **ls –l > outfile**: a legal line of input containing a single token group. In order, the tokens are command, argument, output redirection operator, and file, respectively.
- **ls –l > infile1 | infile2**: an illegal line of input because an output redirect operator cannot be followed by a pipe operator.
- **sort -l < infile | grep hi**: a legal line of input containing 2 token groups.
- **sort < infile | grep hi | wc**: a legal line of input containing 3 token groups.
- **sort < infile | grep hi > outfile**: a legal line of input containing 2 token groups.
- **sort < infile | grep hi | wc > outfile**: a legal line of input containing 3 token groups.
- **> outfile**: an illegal line of input because it does not begin with a word.
- **sort < infile1 < infile2**: an illegal line of input because an input redirect operator cannot be followed by another input redirect operator.
- **ls > outfilel > outfile2**: an illegal line of input because an output redirect operator cannot be followed by another output redirect operator.
- **ls >> infile**: an illegal line of input because two output redirection operators cannot be next to each other. **Note** the command is legal in Linux, which is to append result to the end of the file, but illegal here for the project.
- **sort << infile**: an illegal line of input because two input redirection operators cannot be next to each other.
- **ls > file1%**: an illegal line of input because it contains an illegal character.
- **sort < infile > outfile**: a legal line of input containing 1 token group.
- **ls > outfile < infile**: an illegal line of input because the output redirect operator cannot be followed by an input redirect operator.
- **ls | grep –i hi | sort –k2 | uniq | cut –c4**: a legal line of input containing 5 token groups. In order, the tokens in this line are: command, operator, command, argument, argument, operator, command, argument, operator, command, operator, command, and argument, respectively.

(3) Interpreting the Shell Language

Only legal lines of inputs (as defined in the previous section) should be interpreted. When the shell encounters an illegal line, it prints an error message and then continues to the next line of input.

- Every **command** except the special command **exit** is to be interpreted as a Linux executable to be executed.
- When the shell encounters the command **exit**, it terminates the program.
- All commands are assumed to be in the current directory.

- The ">" operator indicates that **STDOUT** of the associated command should be redirected to the Linux file following the operator. This file should be created if it does not exist (using **O_CREAT** as the flag for the **open()** system call), and the shell should report an error message if the file cannot be created. **Note** that all the contents of the file will be erased if it already exists at the time when it is opened. Similarly, the "<" operator indicates that **STDIN** of the associated command should be redirected from the Linux file following the operator. The shell should report an error message if this file cannot be opened for reading. Make sure all the files opened are closed using the **close()** system call right before your program is terminated.
- The pipe operator (*i.e.* |) indicates that **STDOUT** of the preceding command should be redirected to **STDIN** of the following command.
- After interpreting a command, the shell should wait for all forked processes to terminate before parsing the next line of input.

The following is the interpretation of the legal example commands from the previous section.

- **ls -l**: The shell forks a new process to execute the **ls** command. The main shell process waits for **ls** to exit before reading another line of input.
- **ls -l > outfile**: The shell opens the file **outfile** for writing, forks a new process, redirects **STDOUT** of the new process to **outfile**, and then executes **ls** with **–l** as an argument.
- **ls | grep -i hi | sort -k2 | uniq | cut -c4**: The shell sets up the pipes, forks 5 processes, and then each process executes the command for one of the 5 token groups.

(4) Getting Started

Be sure you understand this project before starting to write code. Here is a rough outline of steps you might take in solving it.

(4.1) Command-Line Parsing

Parse each input line into an array of strings.

(4.2) Interpreting Shell Commands

You will need to use the **fork()** and **execvp()** functions (or variants) to spawn child processes and execute them. The shell process should wait for its children to complete by calling **wait()** or **waitpid()**.

Remember to practice **incremental development**: get something working, add a small feature, test your code, and then move on to the next feature. For example, you should start out supporting shell input that contains a single command and no redirection. Then, add one of these features and then the other.

To support **I/O redirection**, modify the child process created by **fork()** by adding some code to open the input and output files specified on the command line. This should be done using the **open()** system call. Next, use the **dup2()** system call to replace the standard input or standard output streams with the appropriate file that was just opened. Finally, call **execvp()** (or variants) to run the program.

Pipes are a little trickier. You should use the **pipe()** system call to create a pair of pipe file descriptors **before** calling **fork()**. After the **fork()** both processes will have access to both sides of the pipe. The reading process should close the write file descriptor and the writing process should close the read file descriptor. At this point each process uses **dup2()** to copy the remaining pipe descriptor over **STDIN** or **STDOUT** as appropriate.

To help you do the project, sample programs for I/O redirection and pipes are available in the folder of "Resources" on **Collab**.

You should now have a good operational understanding of the user-mode side of some of the most important Linux system calls.

(4.3) Other Odds and Ends

(4.3.1) Code Size

Your shell code should be under 400 lines. If your shell is a lot longer than this, you are probably doing something wrong.

(4.3.2) Getting Help

Since this is an upper division computer science course, you are expected to do your own research regarding the usage of various system calls, header files, and libraries. Information is readily available in the **man** pages, Linux reference books, and on the web. For example, on any Linux machine **man pipe** will give you information about the **pipe()** system call. Otherwise, do not hesitate to ask questions if you are unclear about how some part of the assignment is supposed to work.

5. Grading

Only one codebase should be submitted per group. However, each individual in the group MUST still submit in **Collab**, so that we know you have finished. But your submission in **Collab** will simply be identical for both partners, plus tell us who your partner is. Failing to submit your project to **Collab** will result in losing all the points for the project.

Remember to start early; it's easy to get hung up on problems that would seem easy if you had more time to read documentation and think about them.

Your shell will be graded on the virtual box VM running Debian. So you must make sure that it compiles and runs properly there.

You should hand your project in *via* **Collab** a *tar file* with all of your sources, headers, a **make** file, and a write-up including the source files and description about how you design and implement the solution for the project (at least 2 pages long).

The *tar file* should be named as follows: **p1**, followed by the first letter of your first name, followed by your last name, and followed by the file extension (*i.e.* **tar**). For example, the tar file turned in by **John Smith** should be named **p1jsmith.tar**.

For those who are not familiar with creating a **make** file, the following link will help you understand how to do it: http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

The following shows the criteria we will use to test and grade your projects.

(1) Programs failing to compile will lose all the points.
(2) Each of the examples shown under section (2.1.3) will be used as test cases to test you programs (including error message generated from each test case). This is the major part of points you will receive (18 points).
(3) Description of how you design and implement your projects will be examined and evaluated (2 points).
(4) Make sure your make file is created in such a way that it contains correct directives to initiate the action to compile and run your program. Failing to do so will cause you to lose 3 points.
(5) Regrade is acceptable only in a week from the time when your grades are posted on **Collab**.