**CS6456 (F2015) Operating Systems**

**Project 2**

**Title: Writing Your Own Shell**

**Due: September 21, 2015 (11:00 a.m.)**

**Points: 15**

(1) The Problem

Write a simple Linux shell 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. It is fine to

discuss the specification and its interpretation with other students, but do not share code. 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 100 characters. If a line containing more than 100 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 100 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, forward slash, 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.
* The only legal input to the shell, other than lines consisting of valid tokens, is the end-of-file, which will terminate your shell program (The other command to terminate your program is **exit** which can be entered at the prompt, see below).

(2.1.2) Parsing the Shell Language

* Lines of input are divided into token groups. 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

* 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 one or more file redirections. A file redirection consists of one of the operators < or > followed by a single word called the filespec. 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 OR they may contain an input file redirection, OR neither, but NOT both.
* 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

* **/usr/bin/emacs**: a legal line of input. It contains a single token group containing a single token, which is a command.
* **/usr/bin/emacs|**: 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).
* **/usr/bin/emacs |**: 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 > foo**: a legal line of input containing a single token group. In order, the tokens are command, argument, operator, and filespec, respectively.
* **ls –l > foo1 | foo2**: an illegal line of input because the token group is both followed by a pipe and contains an output redirection.
* **> foo**: an illegal line of input because it does not begin with a word.
* **ls >> foo**: an illegal line of input because two output redirection operators cannot be next to each other.
* **ls << foo**: an illegal line of input because two input redirection operators cannot be next to each other.
* **ls > foo%**: an illegal line of input because it contains an illegal character.
* **grep hi < infile.txt > outfile.txt**: a legal line of input containing a single token group, an input redirection operator, and an output redirection operator.
* **grep hi infile1.txt > outfile.txt < infile2.txt**: an illegal line of input because the input redirection operator and output redirection operator are out of order.
* **grep hi < infile1.txt < infile2.txt**: an illegal line of input because two input redirection operators cannot exist in a single line of input.
* **grep hi infile.txt > outfile1.txt > outfile2.txt**: an illegal line of input because two output redirection operators cannot exist in a single line of input.
* **grep hi myfile1.txt < sort myfile2.txt**: an illegal line of input because the input redirection operator should be followed by an input file, not a command.
* **grep hi myfile1.txt > sort myfile2.txt**: an illegal line of output because the output redirection operator should be followed by an output file, not a command.
* **ls | grep –i hi myfile1.txt | sort | uniq myfile2.txt | cut –c 5 data.txt**: a legal line of input containing 5 token groups. In order, the tokens in this line are: command, operator, command, argument, argument, argument, operator, command, operator, command, argument, operator, command, argument, argument, 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 either the command **exit** or the end-of-file, it terminates with a return code of zero.
* When a command begins with the character '/', it is an **absolute** pathname and should be executed as-is. All other commands are **relative** and should be assumed to be in the current directory (i.e. **CWD**), and therefore the shell must prepend the name of the **CWD** to the command before executing it.
* The arguments to a command should be passed to the **exec** call in **argv**; **argv[0]** should always be the same as the string that is being executed, with actual arguments passed in slots one and higher.
* The '>' operator indicates that **STDOUT** of the associated command should be redirected to the Linux file named by the filespec belonging to 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 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 named by the filespec. The shell should report an error 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. Also, after all forked processes have terminated, the shell should report their return values (e.g. one per line in the order that the commands were specified in the line of input).

The following is the interpretation of the legal example commands from the previous section. Assume that the **CWD** is **/home/john**.

* **/usr/bin/emacs**: The shell forks a new process and in it executes **/usr/bin/emacs**. The main shell process waits for **emacs** to exit before reading another line of input.
* **ls –l > foo**: The shell opens the file **foo** for writing, forks a new process, redirects **STDOUT** of the new process to **foo**, and then executes **/home/john/ls** with **–l** as argument one.
* **ls | grep –i hi myfile1.txt | sort | uniq myfile2.txt | cut –c 5 data.txt**: 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 assignment 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

Modify the shell command loop to properly interpret lines of input. You will need to use the **fork()** and **execvp()** (the system will search the path automatically for you) or **execve()** (if you know how to set up the executable path using the PATH environment variable) functions to do this. The shell process should wait for its children to complete by calling **waitpid()**. The shell should also check and report the exit code/status returned by the program. Note: the exit code is placed into the lower 8 bits of the status code set by **waitpid()**.

The full version of your code will look roughly like this:

while (1) {

read a line of input;

parse the line;

for each command in the line {

pid = folk();

if (pid ==0) {

do redirection stuff;

execvp(command, args, ….) or execve(command, args, ….);

else

store pid somewhere;

}

}

for each command in the line {

waitpid(stored pid, &status);

check return code placed in status;

}

}

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 relative pathnames, use the **getcwd()** system call to find the name of the directory in which your shell is running. You can make this call once at startup time since your shell does not need to support changing directories.

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 **exec()** 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 500 lines. If your shell is a lot longer than this, you are probably doing something wrong.

(4.3.2) Getting Help

Since this is a graduate 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

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. You should also hand in a printed copy of your write-up in class.

The *tar file* should be named as follows: **p2**, 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 **p2jsmith.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 not get any points.

(2) Programs failing to parse input commands will lose a majority of points.

(3) Each of the examples shown under section (2.1.3) will be used as test cases to test you programs (including the exit code returned and error message generated by each test case). This is the major part of points you will receive.

(4) Comments, indentation, meaningful variable names used, blank lines inserted, etc. will also count.

(5) Description of how you design and implement your projects will be examined and evaluated.