CS 4414 Operating Systems – Fall 2017

Homework 1: Writing a Simple Shell

Jonathan Colen

Problem Description:

The goal of this assignment was to implement a simple Unix shell that could run programs, handle file redirection, and manage pipes. This assignment was completed and the program runs correctly.

Approach:

The program operates by running the **shell** method, which contains an endless while loop. At each iteration, a line is taken from standard input, parsed into a sequence of token groups, and executed. The input is handled using the **fgets** command. The user can enter as many commands as they desire and can end the program at any time using the "exit" command. If a file has been redirected to the standard input for the shell program, then the program will continue reading lines until the end-of-file marker has been reached.

Each line is parsed and stored a structure called **token_group**, which will be referred to as **Group**. Each Group contains the command to be executed and its arguments, file redirections, and pipe file descriptors. Token groups that are connected via a pipe will have pointers to each other stored in their Group structure. In addition, each Group has a place to store the process id and exit status of the resulting process for management purposes.

Input lines are parsed using the method **parse_line**. This method separates the input string into a sequence of tokens separated by spaces using **strtok**. Next, it creates a new Group and sets the command of that Group as the first token in that line. All subsequent tokens before the next occurrence of the pipe symbol ("|") are stored in a list string for later parsing. When the pipe symbol is reached, the list of associated tokens is processed and a new token group is created and connected via a pointer to the previous token. This is repeated until there are no more tokens in the line.

The associated tokens for each command are processed using the **finish_group** method. The tokens are checked for the file redirection operators ">" and "<". If they exist, the subsequent strings to each operator are stored in the Group structure as the input or output filenames. In addition, their compatibility with the existing pipes for the token group are checked. If the token group already has an associated input pipe, then any input redirections will be disallowed. Similarly, output redirections will be disallowed if an output pipe exists. All other associated tokens are stored in the string list storing the arguments of the Group.

After the line has been parsed, the **execute_line** method is called with a pointer to the first token group of the line set as an argument. This method runs through a while loop where at each iteration, the current Group is executed and the pointer is incremented to the Group associated with the current command's output pipe. The loop iterates until there are no more commands to be executed.

Execution of a Group is done in several steps. First, if the Group has another process that it should be piped *to*, a new pipe is created and it is set as the output pipe for the current Group and the input pipe for the current Group's successor. This means that no Group ever has to create its own input pipe. This method of creating pipes was chosen because the first Group in a line could not have an input pipe due to the constraints of the language. Thus, the Groups were made to simultaneously manage their own output pipes and their successor's input pipes in order to allow each Group to be handled in the exact same way.

Next, a child process is forked. The child process either closes the write end of its input pipe and sets its standard input as the read end of that pipe, or it opens a file with the name stored during the line parsing and sets standard input as the resulting file descriptor. If there is no input pipe and no

redirected input file, then nothing is done and standard in is left unchanged. Similarly, the read end of the output pipe can be closed and the write end can replace standard out, or a file can be opened and set as standard out if the Group structure contains an output file redirection. Finally, the command is executed using **execve**. If the program fails to execute, an error message is written to the console.

Meanwhile, the parent process simply stores the process id returned by **fork** into the Group structure for later access. Next, the Group structures are iterated through again and the parent calls **waitpid** on the process ids stored for each Group. The returned status is stored in the Group and the write ends of the output pipes and read ends of the input pipes are closed. It was found that neglecting to close these pipe ends would result in the overall shell failing to terminate correctly due to processes continuing to run. After all processes have finished executing, their exit codes are printed out by once more iterating through the Group structures. Finally, the Groups are deallocated and another input line is accepted.

Results:

The program executes as expected and all specified functionalities have been implemented. If an input line is invalid based on the language specification supplied, an error message will be output with the form 'Command "*command_here*" is invalid'. The program also handles incompatible redirections and pipes to nowhere. Relative file paths are supported if the filename does not begin with the forward slash character. If a process fails to execute, a message is printed to the standard error console, and otherwise, the exit code is printed to standard output.

All forked processes terminate as expected. This was verified by running the **ps** Unix command in a terminal and checking the output while the shell program operated.

Analysis:

Parsing input strings was the most difficult and involved portion of this assignment. The initial attempt involved tokenizing the string and then assigning each token a numerical flag to designate it as a "command", "argument", "operator", or "file descriptor". The idea behind this was that execution of a line would require using the flags to link arguments and redirections to their associated commands. This approach became very difficult to manage and was abandoned quickly.

The second approach, which is the one described above, is the result of planning the string parsing with actual execution in mind. I realized that the goal of the program was to execute commands, and therefore the goal of the parsing should be to create a structure for each line that allowed for easy execution of the commands stored in that line. The parsing subroutine then shifted from one that attempted to flag and link tokens to one that sought to fill up a structure. This method made many later problems much simpler to solve. Checking compatibility of pipes and file redirections now only requires checking whether two pointers in a Group structure are NULL. The filenames for input and output redirection can be accessed instantly for a given command. The Group structure also allows for easier process cleanup, as the process ids and statuses are stored in each Group as well.

It should be noted that each Group structure stored its own input and output pipe information. Because one Group's output pipe is another Group's input pipe, this means that there is some data redundancy. This was a conscious decision intended to allow each token group to be independent. For cleanup purposes, it was simpler to associate each command with its "own" pipes. This could also have been accomplished by using a global pipe array that could be filled and accessed by all processes. It was ultimately a stylistic decision to put the pipes in the Group structures, as it kept all necessary information for each process contained within a single structure.

Utilization of data structures simplified both the parsing and execution stages of the program and saved hours of work. They provided significant organizational advantages and made many of the process management procedures very straightforward.

Conclusions:

The shell program operates by accepting input lines, parsing those lines to fill token group structures, and executing the commands associated with each token group. Using data structures to represent commands and token groups simplified the parsing and process management. The program properly utilizes the **fork(), exec(), open(), dup2(),** and **pipe()** system calls. All facets of the specified language are supported and the program executes commands as expected.

Code:

The project is stored in jc8kf.tar
The code is contained in **shell.c**
The code can be compiled by running **make**
The program can be run by executing **./msh**

Pledge:

On my honor as a student I have neither given nor received aid on this assignment.