Luke Grammer
CSCE-313
3/02/19

Programming Assignment 3 Analysis - UNIX Shell

Project Design

My project design resides entirely in the 'Interpreter' class, which contains all the functions required to process shell input, produce a prompt, and manage background processes. The main function (in shell.cpp) creates an Interpreter instance and begins an infinite loop. In this loop, the program parses an input string into a vector of commands, where each command is itself a vector of strings. The elements in each command consist of whitespace-separated tokens, string literals enclosed with single or double quotes, or operators like '<' or '&'. Commands themselves are separated by piping operators ('|') that are not enclosed in a string literal. After the input is parsed into this two-dimensional vector, piping is handled if necessary, and each command is parsed and executed separately.

Piping

Piping is handled in the 'pipe_commands' function in the Interpreter class. This step is only done if at least one piping operator ('|') was found in the user input (if commands.size() > 1). In this step, a copy of the original input/output file descriptors are made and then a for loop iterates through every command. The input of each command after the first is retrieved from the output of the previous command using a pipe. The loop goes on to parse and execute each command, piping the input and outputs appropriately. When the last command is reached, the output is assigned the copy of the default output file descriptor. After the loop completes, the default I/O are reinstated by their copies made at the beginning, and the fds of the copies are closed.

Redirection

Redirection is handled when parsing each individual command so that redirection can be done within pipes. Two fds are created as local variables and initialized with the value zero. When parsing commands, each command token is searched up to the second to the last token for the '<' operator. If one is found, the filename in the next token is recorded and opened for reading with one of the fds being its file descriptor. The command is then searched for the '>' operator similarly. When the program forks to execute the process, if the child sees that the fds are nonzero, it will use the dup2() function to redirect it's input/output appropriately. Since the '<' and '>' operators are searched and processed separately, the commands can be used together

with both input and output redirection (i.e. ‘grep bash < test.txt > test2.txt’), more closely emulating the behavior of other bash shell implementations.

Background processes

When each individual command is parsed, the program checks to see if the last token is ‘&’. If it does, the program sets a flag which is then checked in the parent process after the child has forked and executed. If the flag is set, instead of waiting on the child process, the parent prints the pid of the created child and adds it as a ‘Process’ object to a list of background processes. To avoid producing zombie processes, this list is checked after every command in the ‘manage_processes’ function to determine which processes have terminated, and if they have, a message is printed after the next command is entered and they are removed from the list of running processes. If the ‘exit’ command is entered while processes are running in the background, a ‘shutdown’ function is called in Interpreter to wait for the processes to complete before the shell is terminated.

Directory Management

The directory management is done primarily in the ‘execute_special_command’ function of the Interpreter class. The class maintains two member variables for the current working directory (‘cwd’) and the previous working directory (‘pwd’). The pwd is initially the empty string, while the cwd is initially set to the value of the $HOME environment variable in the system using the getenv() function. If the ‘cd’ command is used by itself with no arguments, the cwd is again assigned to $HOME. If there is more than one argument, if it starts with ‘~’, the ‘~’ is replaced by $HOME. If it starts with ‘.’, it is checked to see if it is ‘..’. If it is not, the ‘.’ is removed and ignored since it references the current directory, and if it is, the directory path up to but not including the current directory is appended to the path argument. If the command is ‘-’, cwd and pwd are swapped, otherwise, the directory specified is navigated to and changed if it exists (absolute paths begin with ‘/’ while relative paths do not).