EECS 678 - Project 1 [Quash Shell] Report

Author: Jace Kline (KUID 2881618)

Goal and Overview

The purpose of this project was to implemented a limited-functionality shell mimicking the behavior of BASH in preferably the languages of C or C++. This project required general programming expertise, the ability to use POSIX system calls, and the ability to use concurrent programming, namely forking, piping, and redirection. I chose to implement this project in C++.

Features

Parsing

One of the biggest challenges in implementing this project was parsing the relatively complex syntax structure of the possibilities of inputs that could be provided to the shell. To parse the input, I followed the general sequence of steps outlined below:

1. Extracted the first chunk/word. If it is a built in command, then attempt to construct a Builtin object. Otherwise, attempt to construct a Pipeline object.
2. The Builtin object type was simply a wrapper around a vector of strings (command + argument list).
3. The Pipeline object type was parsed as follows...
   - Determine whether to run in background ('&' character at end)
   - Split the remainder string into chunks, delimitted by the pipe '|' character
   - For each of those chunks, split up based on either whitespace or matching quotation marks. This leaves me with a vector of vector of strings.
   - Attempt to find the redirection ('<' or '>') operators and extract the filename that they are associated with
   - Ensure that the same redirection operator does not appear more than once
   - Iterate through all string chunks and invoke failure if a special/meta character is found as a standalone
   - If none of the prior steps failed, construct the Pipeline object and return
4. The result of the parsing procedure is strored in a ParseStruct object, that essentially acts as an either-or type for the Builtin and Pipeline types. This information is then passed to the ExecutionEnvironment object to be executed.

Environment

To keep track of the environment, I created a class (Environment) that acted as a pseudo-Environment for the program. It handled and updated the HOME, PATH, and current directory of the Quash process. Because the ExecutionEnvironment object has a member variable of the an object of this Environment class, all forked processes from the ExecutionEnvironment's 'execute' method would automatically gain access to the Environment object. For other environment variables, I simply parsed the input and used the 'setenv' POSIX system call.

Filepath Resolution

To challenge myself, I made a miniature recursive parser for expanding relative filepaths when given a string and a working directory. This was very useful when it came to expanding paths for creating argument arrays to pass to the 'execv' system call in the child processes. The general idea was that it is essentially a "fold" from left to right, with each recursive call being passed the resolved path from the previous step as the working directory. Additionally, I created functions that determine whether a given file is an executable or a directory. These came in handy when checking executable paths in the Pipeline chain.

Background Job Handling

Handling and keeping track of background jobs was an important feature in the Quash shell. To achieve this, I created a class called a JobHandler that stored a vector of Job data structures (structs) and an incrementer integer to generate unique job ids. Each Job stored the job id, process id, and command name. The process id of a Pipeline chain would be stored as the process id of the last process in the chain. On each main loop of the shell, I called the 'refresh' method of this class that would use waitpid(..,..,WNOHANG) to get the status of the process id. If any completed, the method would print and the Job would be removed from the vector.

Execution

The most important part of any shell is the execution of the commands. In my Quash implementation, I used a ExecutionEnvironment class to represent the "executer" of parsed commands. The member variables of this class were an Environment object and a JobHandler object. These are what essentially stored the state of the shell, and therefore it was important to have them in scope during execution. When the 'execute' method was passed a ParseStruct, either the ParseStruct object held a type Builtin or held a type Pipeline. Based on the type, the corresponding 'executeBuiltin' or 'executePipeline' methods were run. The Builtin executer would simply look at the first command and branch to that condition, then it would ensure valid arguments and argument list length before attempting to execute the functionality of each command. The Pipeline executer was essentially a loop through all pipe commands where it would (if not last command) create a pipe, overwrite the stdout file descriptor (1) to the write end of the pipe, and reassign the input file descriptor to the read end of the pipe for the next iteration. Special cases occurred when the command was the first or last in the command sequence. In these cases, it was checked whether there was a redirect in/out file and overwrote the correct file descriptor accordingly. Lastly, if the command was run in the background, then a Job object must be created and stored in the job handler. If the command is not run in the background, then the last process in the chain must be waited on with waitpid(...).

Testing

To perform testing, I simply created input files of sequences of various different commands and fed them into Quash. I also tested all of the test files provided on Blackboard. During development, I would unit test particular features/functions to ensure they were performing as expected.

Shortcomings

Although my implementation of the Quash Shell works well, I wish I could add/change the following:

- Not use exceptions to generate error messages
- Add the ability to run builtin commands within a pipeline chain
- Have a more robust and efficient parsing system that is reusable and extensible
- Have a more clear and consistent class/struct hierarchy with more separation of responsibilities
- Not use the 'setenv' system call, and instead implement an entire pseudo-environment