EECS 678 - QUASH Report

Samuel Lamb
Theodore Lindsey

March 8, 2015

1 Intro

QUASH is a LINIX shell terminal similar to bash or csh. It uses a command-line interface to execute commands. Our approach to implementing QUASH is essentially outlined as follows:

1. Gather user input
2. Check if input corresponds to a built-in function
3. Determine if pipes or stream redirection is necessary
4. Tokenize raw input based on presence of pipe or stream redirection characters
5. Parse tokenized strings into a custom data structures for ease of access
6. Pass parsed strings off to execution functions
7. Go to 1.

2 Pipes

We needed to consider two cases involving pipes. Either pipes only or pipes with stream redirection. In either case, we tokenize the user input as necessary and then pass it on to our execution functions. If it is just a pipe we tokenize the user's raw input based on the '—' and newline characters. If both stream redirection and pipes are present, we first tokenize the raw input on the '—' and newline characters and then on the stream redirection characters at which point we simply pass the corresponding strings off to our execution functions. Within the execution functions, prior to replacing the processes image, pies are initialized and remapped to and from `STDIN_FILENO` and `STDOUT_FILENO` at which point process image replacement occurs as normal.

3 Stream Redirects

Since the case when stream redirection and pipes are both involved was already addressed, we just need to consider the case of only stream redirection. In this case, we tokenize raw input with the stream redirection characters and then pass the necessary strings to the corresponding execution function. We then redirect `STDIN_FILENO` and `STDOUT_FILENO` to and from files as necessary at which point process image replacement occurs as normal.

4 Command Execution

We begin by identifying if there are pipes or data stream redirections in the user's input. Once that has been determined, we break the users input up around the pipe symbols and data redirection symbols. From there, we parse each component string into a command, its arguments, and a few other elements of data and, based on if pipes or stream redirections are called for, pass the necessary data structures into one of several executing functions. From there, we fork the process and in the child, we replace the current process image with that of whatever the user entered. This occurs within the necessary code support in order to facilitate a pipe or stream redirection (as necessary). If the user didn't request that the command be backgrounded, the parent thread waits for the child to terminate and then returns back to waiting for user input.

5 Built-In Commands

- `pwd` - prints working directory, no arguments
- `cd` - changes directory, defaults to home directory, otherwise specified directory
- `$PATH` - displays the path environmental variable
- `$HOME` - displays the home environmental variable
- `set` - sets home and path environmental variable with syntax `set PATH=...` or `set HOME=...`
- `clear` - clears the screen
- `exit` & `quit` - exits the shell

All environmental variables are checked for and parsed prior to checking for other commands. The raw input from the user is compared to the characters of the command. As the back end implementation for each is quite simple, all except `cd` are implemented in the main function and just strip down the raw user input until they have the necessary argument assembled.

6 Problems Encountered

The two most problematic issues we ran into were with the way that C handles strings and struggling through the documentation of functions that seemed to be essential to the implementation of the shell. Due to the limited functionality surrounding the string implementation, parsing user input was quite tricky. Rather than, for instance, simply running the user input through a regex search, we instead needed to break down the input with tokenization into small pieces. Unfortunately the tokenization resulted in naive parsing that couldn't account for complex situations (such as mixing input redirection, pipes, and backgrounding). We suspect that had we used a higher-level language with more extensive data types, the task would have been much more approachable.