QUASH

"Quite a shell!"

Gehrig Keane and Joseph Champion

EECS 678 Project 1 | March 2, 2016

Introduction

QUASH (QUite A SHell) is a text-based user interface providing access to services from the Linux operating system. Its functions and user experience are modeled after that of GNU Project's shell, Bash.

Features

QUASH was designed to support the following terminal commands and functionality:

- Run executables, with or without associated arguments;
- Provide an option for executing foreground and background processes;
- The **set** command, for assigning values to the operating system's environment variables for $HOME and $PATH;
- The **cd** command, enabling the user to move the current working directory to a specified location on disk;
- The **echo** command, for displaying the values for $PATH and $HOME;
- The **pwd** command, to display the absolute path of the user's current working directory;
- The **jobs** command, to list all currently-running background processes;
- The **kill** command, which kills background processes when provided with a kill signal number and job identification number;
- Leave the shell instance with either an **exit** or a **quit** command;
- Process a batch of QUASH commands from a file;
- Redirect the input or output for running processes to or from a file, or pipe the results from one process into another.

Implementation

As mentioned previously, QUASH shares similar behavior and command-line syntax to Bash. The project was coded in C, using the GNU99 compiler mode.

QUASH and its core functionality lies in parsing commands from an interactive, bare-bones shell instance. Command interpretation begins with tokenization in similar fashion to traditional C *args (see the **get_command** function). Tokenization enables flexible execution with the Linux system in that commands may stem from any general filestream, notably standard input. After parsing a command, QUASH begins shipping the tokenized structure through its core decision structure.

get_command (command_t* cmd, FILE* in)

QUASH continually iterates on **get_command** providing the user a traditional terminal experience. **get_command** tokenizes commands into a form similar to that of argv, except commands are stored within the **commant_t** struct:

```
typedef struct command_t {
        char** tok;
        char cmdstr[MAX_COMMAND_LENGTH];
        size_t cmdlen;
        size_t toklen;
} command_t;
```

Where **cmdstr** is the raw command string, **cmdlen** is the length of the raw command, **tok** is the array of tokenized arguments, and **toklen** is the amount of tokens in the **tok** array. By storing each command issued to the QUASH terminal within the **command_t** struct, it becomes much easier to pass a packet of key information between each piece of the internal QUASH decision structure.

run_quash (command_t* cmd, char envp)**

The first point of the execution process, **run_quash** acts as a simple decision engine for the redirection of command execution towards an appropriate function below. The necessity for **run_quash** stems from manual implementations of several traditional linux programs. Namely, cd, set, etc… required internal implementation establishing the need to separate the execution of these commands from existing linux system programs.

exec_command (command_t* cmd, char* envp[])

This function acts as the second layer for the decision engine initiated by **run_quash**, which uses a rudimentary parser to further direct control flow for executing commands. Specifically, the control flow for **exec_command** is as follows:

Symbol encountered	**Method invoked**
&	**exec_backg_command**
< or **>**	**exec_redir_command**
|	**exec_pipe_command**
Not applicable	**exec_basic_command**

exec_basic_command (command_t* cmd, char* envp[])

Commands without any of the unique identifiers mentioned above fall into the basic command category, therein these programs are likely located in the current working directory or within one of the environment paths. Such programs are very easy to execute with the **execvpe** system call. Note **execvpe** passes the environment variables caught by main as an argument to the child process allowing the inheritance of environment variables. This development decision, which carries throughout the remaining execution

functionality, satisfies the abovementioned rubric criterion. The implementation of **exec_basic_command** was relatively straightforward in that creation and management of a child process required no more than handing the command off to the system.

exec_redir_command (command_t* cmd, bool io, char* envp[])

As stated above, commands which contain a less than or greater than sign are sent to the **exec_redir_command** function. The motivation for discretizing the functionality of a redirection command stems from the need to manipulate the input or output streams of the child process involved. Therefore, in **exec_redir_command** we either map STDIN or STDOUT to the appropriate file descriptor. Otherwise this execution function differs very little from that of **exec_basic_command**. Notable implementation struggles included the syntactic structure of child functionality and research for pertinent system arguments.

exec_backg_command (command_t* cmd, char* envp[])

This execution function coupled with the following, provided the most struggle with regard to implementation and elimination of fragility. The **exec_backg_command** initializes a child process and stores the specifics of said child process within the following struct:

```
typedef struct job {
        char* cmdstr;
        bool status;
        int pid;
        int jid;
} job;
```

Where **cmdstr** is the title of the background process to be created and inherits directly from the **command_t cmdstr** attribute, **status** is used to mark jobs that are either alive or have finished, **pid** is the process identification number for the created child process, and **jid** is the QUASH job identification number. This structure allows easy listing and termination (with the kill function) of background process created. As mentioned above, innumerable struggles during the implementation of the function at hand causes major headache and the current implementations suffer from several flaws. Namely, jobs created in the background aren't provided any functionality to for retrieval and placement in the foreground, meaning they cannot collect user input unless provided in the original arguments chain. Also the current implementation of background and redirection processes is disjoint. One notable achievement lies in the mapping of the **job_handler** function to child process termination allowing responsive jobs struct maintenance and the typical background process termination message " [0] 12345 finished <some command string> ". The **kill_proc** function also interacts with jobs placed in the job struct array, allowing the QUASH user to kill various background processes with a kill signal and jobs identification number.

exec_pipe_command (command_t* cmd, char* envp[])

The pipe command allows the QUASH shell user to create system pipe and rope the STDIN and STDOUT streams of programs together. The implementation should function with any number of pipes though the commands within the pipes are subject to fragile restrictions. Notably, arguments are delineated in the tokenization process by spaces therefore if say an awk command is issued with {print 1$} execution will fail due to tokenization practices. Furthermore, piped commands aren't currently compatible with background or file redirection commands. While the implementation of such functionality was initially planned, development hiccups usurped said plans.

exec_from_file(char argv, int argc, char* envp[])**

This function enables QUASH to support batch task execution by reading commands from a file. When QUASH is initialized, it queries the source of any input arguments with **isatty**. If the input for QUASH was not provided by the terminal (via typed commands), then it is assumed that the input stream was provided by a file. QUASH will then use **exec_from_file** and attempt to parse file contents and execute tasks in identical fashion to terminal input. Since **isatty** is reached before **run_quash**, this function is given a chance to execute before the command string was parsed by **run_quash**. Thus, any QUASH commands read from a file using the syntax **./quash < commands.txt** will not be incorrectly handled by **exec_redir_command**.

Conclusion

QUASH provides a simple, text-based user interface for running programs and managing files. Each of the commands **cd, echo, exit, jobs, pwd, set,** and **quit** were directly implemented to achieve similar or equivalent behavior to the Bash alternative. By handling all incoming commands with **execvpe**, QUASH is capable of executing other Bash commands that were not directly implemented, in addition to I/O pipes and redirects. Lastly, QUASH supports reading from a file, enabling tasks to be scripted and automated.