QUASH

Version <1.0>
3.7.2015

Roxanne Calderon
Lynne Coblammers

Introduction

Quash (quite a shell) is a command line shell used to implement UNIX system calls. This program is implemented in C and functions similarly to bash. Information on initiating the program can be found in the README.md file.

Taking in Arguments

The most basic functionality of the program is to take in arguments and parse them in a way that can be easily understood by the rest of the program for execution. In quash, the current working directory is retrieved and printed with a '>' symbol as a command prompt. The user can then enter a command. Commands are read in and split up to create a command vector (char**). Parsing the command this way provided an array of arguments that could be accessed easily through args[i], similar to how argv is used in main. This allowed the program to easily check the number of arguments, making error checking simple if not enough arguments were provided. As is standard in UNIX system calls, the program would treat the first argument as the primary command.

The command is first read one character at a time into a dynamically allocated char*. This method was chosen to allow for arbitrarily long input. Once a newline character or EOF is read, the string is tokenized with spaces as the delimiting character and stored in a dynamically allocated array of strings (char**), or command vector. The final argument of each command is always set to null, allowing the rest of the program to scroll through the arguments until null without creating a segmentation fault.

Once the commands have been properly parsed and set up in the char**, they are processed. The first argument (the command itself) is compared with each of the locally implemented commands, which include cd, jobs, set and kill. These commands are implemented in functions and are called into from main. If the command does not match any of those, it is processed as a general command.

Set Command

The set command does one of two things. With no arguments, it prints the current values of HOME and PATH, two environment variables. These variables are inherited from the caller of quash and are passed in through an argument to main, envp*. The user can also use the set command to change the value of HOME or PATH. For example, set PATH=/bin:/usr/bin:/user/Lynne/678 will change update the value of PATH. PATH is critical because anytime a command is called without explicitly listing the complete file path, the directories listed in PATH are searched for that executable. This is done by calling excvpe, which passes in the environment variables and searches PATH if the command does not contain '/'. HOME is used in cd command with no arguments, as described below.

Changing Directories
This command could be implemented quite simply with the chdir command which is part of the Unix standard library for c. This command will take in a path (which the user provides as an argument) and changes to it automatically. To ensure that ‘chdir’ works without error, the command first checks to ensure the path is valid, returning an error message if it is not. If no argument is provided, the path will automatically be set to the users home directory.

Jobs Command
The purpose of the jobs command is to display all running job processes, including their job number, their PIDs and the name of the command. This is set up with a struct that contains all relevant information, and a flag for when the process ends. Jobs are set in the parent process of a background command. Each job is inserted into an array of up to 1000 jobs, and a simple for-loop scrolls through all unfinished command when the job command is called.

Kill Command
The kill command requires two arguments: a signal as the first and a process (using the job ID). If either of these commands are not input, the terminal will alert the user. If both commands are correct and the signal is not zero, the signal will be sent to the process and the process will be removed from the jobs list.

General Commands
All other commands are passed to an execCommand function. This function checks all of the arguments to determine if there is an ampersand, denoting a background process, a pipe, denoting piping, or either of the redirection symbols. If one of those is found, the appropriate execute function is called. If none are found, execSimpleCommnad is called, which forks a child process. The child uses execvpe to execute the command, and the parent waits for the child to complete before returning.

Background/Foreground Processes
Foreground is a basic parent/child execution process. A pid is created for a child by forking the process. While the child is running, the parent process will wait, not allowing the user to enter any more commands until the child process has finished.

A background process is executed if the user enters a “&” symbol after the command as a final, separate argument. In this case, the parent will not wait for the child to finish, using the WNOHANG flag in the waitpid command. The information for the jobs command is initialized in the parent before it exits. Since the parent is not waiting for the child, in order to know when the child finishes, a signal handler is set up for SIGCHLD. The child will announce it is running in the background, displaying its job number and pid. The foreground will return to quash and allow the user to continue entering input until the background process is finished. The background process will send a signal, which is passed to the handler. The handler will announce that the child is finished, printing its job number, pid, and command name. The handler also changes the flag of the jobs process to indicate it should not longer be displayed.

I/O Redirection
If an I/O redirection symbol (< or >) is found in the command, it is executed as a redirected command. The final argument to the redirect command is the file to replace the stdin or stdout with. This file is opened in the appropriate mode (read or write only) with appropriate permissions, producing a file descriptor. This file descriptor is then duplicated to either standard input or standard output depending on the symbol. The command is stripped of its last two arguments, and executed as normal by forking a child, calling execvpe, and having the parent wait for the child's completion.

Pipes
If any pipe symbols are found in the command, the first step is to split the command into multiple commands. The splitCommand function is called which takes a single command vector and splits it into a set of command vectors that no longer contain pipes. This set of command vectors is passed to a function to execute piped commands. The function allows for multiple pipes. It begins by allocating the total number of pipes needed (number of commands minus one). Next, it enters a loop that handles one command at a time and continues to loop so long as there are commands. This loop duplicates the stdin and stdout with the appropriate pipe fd for each command. If the command is the first command, only stdout is replaced. If it is the last command, only stdin is replaced. For all other commands, both standard out and standard in are replaced with the correct pipe. Once the input and output have been set up, execvpe is called on that command. Outside of the loop, the parent waits for each child in a loop as well, exiting once all children have terminated.

Reading Commands From File
To determine if quash is being called with a file as input, the first step is to check whether stdin is connected to the terminal. If so, the program proceeds as normal. If not, it is assumed to have been redirected to read from a file, and a separate function for executing commands from file is called. This function first reads all of the commands in the file, separated by newlines, and creates an array of command vectors (char**). These commands are then executed one at a time. Although quash does not print any prompts when reading from file, it still checks to determine what to execute in the same way, comparing the first argument of the command with each of the locally implemented functions. If none match, it calls execCommand and processes the command in the same way as described above.

Conclusion
Quash provides a simple shell environment that is easy to use. It reliably executes cd, set, kill, jobs, simple commands with and without arguments, redirected commands, background processes, and commands with multiple pipes. It also allows a user to create a set of commands in a file and execute those directly. This allows more casual users to easily run executables without having to make system calls or understand interprocess communication.