Isaac Cook
Howard Grimberg
QUASH – Project 1
EECS 678 – Spring 2013

Overview

The general architecture of the quash mirrors many other parsing systems. A rudimentary parser will attempt to parse the inbound string (be it from terminal or a file) and attempt to create a data structure containing a list of commands to execute and everything that is needed to do so for each command. The arguments and environment become self-contained, making execution and job management trivial. Initially, we decided to use pure C to complete the project. However, as the project progressed, we realized the necessity of using C++ and the wide variety of data structures that come integrated into the language.

Execution

Execution is handled by a simple wrapper around exec. The system call execvpe has built-in functionality to search the path if the arguments specified do not have an absolute path, thus eliminating the need to parse the PATH ourselves. Before calling execute, a fork() is called to spawn a child process in the usual manner. The pipes and redirection are set before the execute function but after the fork. The execution command inherits and passes the parents environment (also part of the execvpe system call).

Built-In Commands

Built in commands are handled simply by intercepting the parser before it can handle any other characters. For example, if the line buffer has “exit” the shell will return 0 and exit on the spot. For changing directories, a function change_dir is called and is given the current line buffer. It then determines what arguments (if any) are present, builds a final path and then issues a chdir system call.   The HOME and PATH Environment variables can be changed by using SET. The parser will then attempt to discern what variable is being changed and perform the necessary system call.

Jobs

Job handling is implemented in a very rudimentary fashion. A vector containing the instances of job classes is maintained throughout the execution of the shell. When a child completes, the job listener will fire sigchld_int which will in turn, remove the job from the job list and notify the user. A struct called sigaction dictates what happens on what signal. Every time a new background object is created, it adds a corresponding JobObj entry to the main job vector.

Pipes and Redirects

Pipes and redirects are handled as standard files. As the executor loops over the vector of commands. The loop checks if a command needs to be piped or redirected and takes the appropriate action with pipes and file descriptors depending on where in the list of the commands a given command is. If the command is first, then STDIN must be STDIN and if the command is last its output must be STDOUT unless it is being redirected. Commands in-between the ends have both STDIN and STDOUT connect to pipes. On each iteration, a new pipe is created and connected to the appropriate start and ends points. If the output is being redirected, the STDOUT file descriptor is replaced with a descriptor pointing to a regular file.

The same is true for redirect in, except the STDIN file descriptor is replaced with a file set to read.

Inbound Commands

The shell can handle input using two different methods. The first method is the usual interactive method of asking the user for commands to execute. When this happens, the user and hostname are printed to terminal. The shell also pauses until the user enters input and presses return. The parser begins to read the users input. When redirecting a list of commands in, the same thing happens. C++ has facilities built-in to read files line by line(in contrast with C). Because the list is attached to STDIN, no additional facilities are needed to handle the list of files.

Challenges

The hardest part of the lab(initially) was not the execution and piping, but the parsing. Originally, we decided to use pure ANSI C(because we are thought we were too cool for C++). We lost a lot of time trying to write a parser in C. The standard C library does not include any dynamic data structures which resulted in either a) using static allocation or b) making our own data structures. Doing this proved to be an immense waste of time. Ultimately, we decided to use C++. As a result, productivity increased greatly and we able to complete the project. We also used Clang++/LLVM as our compiler of choice during development because it(unlike gcc) has useful and not verbose error messages. We also used several routines from the perennially popular Boost C++ libraries(namely algorithms and assigns). We only used boost for string processing and data structures and not actual execution.