CS 111 Design Project for Lab 1: Interactive Command-Line

Vishal Kotcherlakota, Justin Ancheta

Abstract.

Our design project is based off Lab 1 and we plan to implement an interactive command line complete with many of the features bash possesses. We used our base from Lab 1 and used the GNU Readline library on top of it. We were able to get a substantial amount of features complete and our design project does have basic functionality we believe vital to an interactive command line interface. Additionally we identify other important features that would have been nice to implement in our design.

Specification of behavior.

Make your shell interactive, with a prompt, and command-line editing, and tab completion, and have it do something reasonable when you type control-c.

A user needs a command line for many tasks related to use of the operating system, namely: running programs, creating and reading files, manipulating directories, and even installing additional software. To that end, a command line interface (CLI) is an invaluable tool. Typically, the command line program starts once a user authenticates with the login program. The login program then starts the CLI, which waits for user input.

Since the CLI is the user's window to the system, it needs to exploit a variety of operating system features, some of which require access to system-level code. However, the CLI itself is a process running in user mode on behalf of the currently running user. Therefore, the design of the CLI must be a gateway for making system calls, and should do minimal processing of commands--the kernel is the ideal place for most of that work.

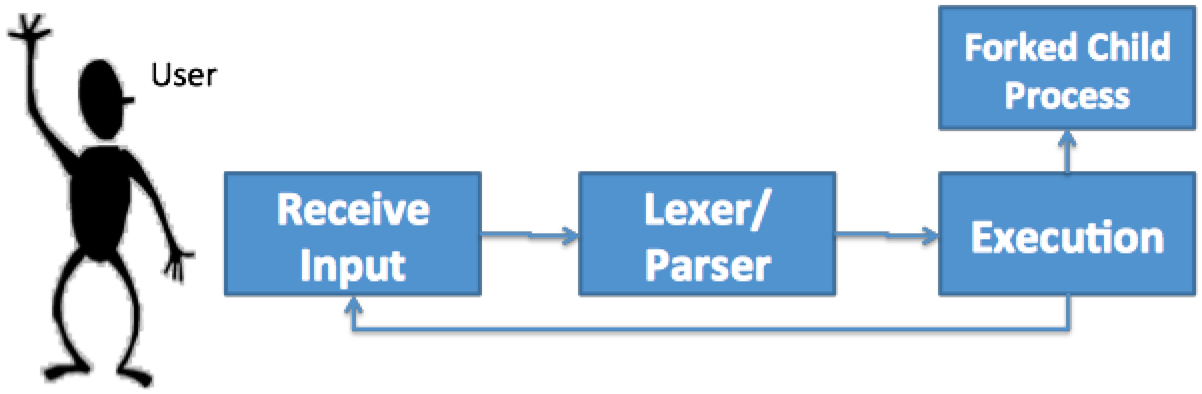
The user should expect the following functionality from the CLI:

- *Interactive Editing.* The user may want some help issuing a command to the system. The CLI can pitch in by allowing the user to edit a command before issuing it.
- *Tab Completion*. Certain programs or files may have very long (but useful) names in the system. To make life easy on the user, we want to be able to suggest potential completions for named objects in a command. This feature has two benefits. First, it saves the user typing. Second, it provides the user with clues as to whether or not an object he is referencing exists.
- *Command History.* Often, a user needs to reference a previousy run command, either to remind himself what actions he has taken, or to run the command (or something very similar) again. To this end, we want the CLI to save commands the user issues, at least for the duration of the session.
- *Syntax/Grammar enforcement*. Like many computer systems, the CLI relies on structured input from the user that follows well-defined rules. However, humans are fallible and forgetful, and need reminders of these rules. As such, when the user issues a command with incorrect syntax, the CLI needs to ignore the command, signal the user with a helpful message, and provide an opportunity to revise the rule.
- *Stability.* Not every program the user runs is stable. To the best of its ability, the CLI

needs to continue running and processing user input even if a program the user runs crashes. (If the OS crashes, we're sunk anyway). If the CLI crashes at any point, the system is as good as dead, since it can no longer accept input.

- *Scripting/Automation*. Many tasks the user may want to perform are complicated and involve invoking many different programs. A user may want a CLI to provide its own scripting language, with support for control structures, so that many tasks can be simplified down to an executable script.
- *Job Control*. At its core, the user's interest is in starting, stopping, and switching between processes (maybe even many of them). To that end, the CLI must support these functions.

Implementation plan.
The figure below shows the basic conceptual workflow for a CLI. The following paragraphs explain each phase in detail.



Receive Input. Arguably, this is the most critical part of the CLI--its facility to gather input from the user. The program places a string of text on the screen, called a *prompt*. The prompt may provide the user with some context on the process (e.g. current working directory, username, etc), and it siginifies the CLI is ready for a string of text from the user, called a *command.* While the user is entering the command, the CLI should provide support for interactive editing--that is, enabling a user to edit a line, copy and paste text, even potentially suggest completions for words the user is typing. After the user signifies that he or she is done entering the command (typically by hitting the Enter/Return key), the input passes onto the next stage. The CLI also needs to be able to remove extraneous whitespace from, at the very least, the beginning of a user's input.

Lexer/Parser. Several transformations need to occur before the user's input to the CLI becomes something the OS can take action on. First, the CLI needs to abstract the users input (a string of characters) into a series of objects with semantic meaning (a stream of tokens). This transformation is known as *lexing* or *tokenizing*. Next, the CLI abstracts the tokens into a set of objects which correspond to things the operating system must do as per the command. This transformation is called *parsing*. Both these steps enforce a set of rules for input (known as *syntax* or *grammar*), which the user is expected to know. If the command input violates any of these rules, the CLI should abort the operation. warn the user, and offer a chance for the user to edit the command.

Execution. Once the CLI finishes parsing the command stream, it now needs to "execute" the commands--that is, pass them to the kernel for execution via system calls. However, not every command that is parsed gets executed. There are conditional primitives (AND, OR) which require special handling. Additionally, not every command takes its input and output from the same places. Operators like pipes and redirects imply that the CLI must do some work before invoking the system call. Each command structrue gets passed to the system via its own process.

Summary of results.
The majority of our design project focused on the “receive input” phase of the project, since the other two phases were completed during the course of the class.

Interactive Editing and *Command History* were actually relatively easy to setup and implement. They are both implemented as part of the GNU Readline library that merely required standard configuration and calling the correct functions.

Tab completion was a bit more difficult to implement because of the dual nature that is expected for command completion. If the first word is to be completed, a command in the PATH or working directory should be used. Any word to be tab-completed thereafter would be in the working directory. The Readline library provides word completion in the working directory out of the box. The challenge was extending that functionality to complete commands on the PATH as well. To do this we first read all directories named in PATH and stored each unique directory path in a linked list. We thought this would be preferential to storing every single executable because it would require a lot of memory with the likelihood that 95% of it would never be needed or used. Additionally, the user would need to re-start the program if new executables were added to the path. Our current implementation searches each directory in our linked list of directories for matches and will complete the current word if there is only one or list all possible matches if there are multiple.

Signal Handling turned out to be more difficult to handle than we first thought. This is because of the dual nature that Control-C can take. While the user is inputting a command or modifying previous commands, we wanted Control-C to cancel the current input, not add it to the command history, and present the user with a new prompt to receive input again. If the user’s input has already completed and the command is now being executed, we wanted Control-C to terminate the currently running process and present a new prompt to receive input. We were able to achieve this dual functionality with the help of sigsetjmp and siglongjmp (from setjmp.h for cancelling current input) and by the nature of parent-child relationships and having the parent process handle a Control-C differently.

Using sigsetjmp allowed us to save the state of our parents execution. When the readline library is invoked with readline(“Prompt”), it does not return until the user is finished (hits return). To cancel the current line, we used siglongjmp in the custom signal handler that we installed. This cause the program to jump to where sigsetjmp was initially called. To allow for repeated use, we called sigsetjmp repeatedly until it was successful (returns 0) so that the next invocation of siglongjmp has somewhere to jump to. This works because siglongjmp returns a value other than zero to sigsetjmp.

Job Control. Handling the killing of a child process was actually much easier than expected. Because of the nature of a parent and child, a kill signal (control-C) is sent to a "process group" that will kill the parent and all of its children processes. All that needed to be done was to print a newline to screen and return the parent to its previous execution state because the parent was waiting on its child(ren) process(es) to complete. The child process is killed and the parent is able to tell that it was killed and goes through its cleanup until it gets back to main where readline is called again so the user can input a new command.

Known Shortcomings. Our implementation, as it stands, is only able to operate correctly if valid characters are used correctly. We were unable to spend the time necessary to fix this. The reason it operates this way is because this behavior was intrinsic to our Lab 1 submission.

Further work.
While our implementation has a fair number of features, there remain a number of additional features a user would want that we did not have time or scope for.

Scripting language. While a scripting language is a desirable feature, implementing it would require a massive expansion of the grammar specification and conversely the lexer/parser stage. Additionally, scripting languages are a mature subject, so unless there was a need for a wildly different syntax compared to an existing shell (e.g. bash, tcsh, etc), there really is no need for one.

Support for additional system calls. Our current CLI implementation uses command objects to formulate arguments for the execvp call. While the execvp call is very useful, it is not the only system call a user may need to take advantage of. For example, the user may need to change directories (accomplished by a call to chdir). Supporting this would either require interpretation in the "receive input" stage (which breaks modularity) or the "execution" stage (which requires an augmentation to our command objects). Both were deemed out of scope for the effort.

Task backgrounding and job control. While the user has the ability to kill the currently running process in the current implementation, that only scratches the surface of what can be done. A user may want to take advantage of a multi-core system and run multiple tasks in the background. Additionally, the user may want to switch between multiple tasks. While most of the functionality is already in the operating system, we did not have time to investigate how to accomplish this.

"On-the-fly" execution. In our current model, the system reads the content from the string into memory, transforms it twice over, then acts upon data structures it has created from input. There are two problems with this approach. First, we are representing the command string twice over in memory, once as a string of characters, and once as a set of structures. Second, for longer, more in-depth commands (and scripts), this method is slow. A performance-oriented approach would read from input as needed, through execution.

*Unix/Linux Signal handling.*There are a number of signals the user can send to a process

(SIGINT, SIGSTP, etc). Any of these signals should be routed to signal handlers within the CLI so that they do not impact the CLI itself, but the processes that the CLI initiates.

Division of labor. We developed most of the code in a “distributed pair programming” style. The two of us used a VoIP solution and screensharing software to emulate collaboration at a single desk. We alternated roles of “driver” (the person writing the code) and “navigator” (the person observing the code, critiquing the implementation, and maintaining the “big picture”). Additionally, we use a hosted (bitbucket.org) distributed version control system (DVCS) to maintain a common, coherent base of our code.

Individually, Vishal focused on integrating and validating the use of the GNU Readline library, and Justin took on signal handling in order to implement our desired behavior.

Presentation. http://www.youtube.com/watch?v=LAb9IQ65blw