

Project 2 Specification

Implementing a Shell

Assigned: September 16, 2009, 11:00am

Due: October 12, 2009, 11:59:59pm

Language Restrictions: C only

Additional Restrictions: `system()` and `exec*()` system calls may not be used, except for `execv()`.

Purpose

The purpose of this project is to familiarize you with the mechanics of process control through the implementation of a shell user interface. This includes the relationship between child and parent processes, the steps needed to safely create a new process, including searching of the path, and an introduction to non-trivial user-input parsing and verification. Furthermore, you will come to understand how input/output re-direction, pipes, and background processes are implemented.

Additionally, this project is an exercise in reverse-engineering. Given a specification describing an expected behavior with an existing implementation, you'll learn how to use the existing implementation to verify the correctness and progress of your solution.

Problem Statement

Design and implement a basic shell interface that supports input/out re-direction, pipes, background processing, and a series of built-in functions, as specified below. The shell should be robust (e.g., it should not crash under any circumstance beyond machine failure). The required features should adhere to the operational semantics of the `tcsh` shell.

Questions

1. What is the difference between an absolute path and a relative path?
2. Why is user input dangerous?
3. Why is background processing useful?
4. What is an environment variable?

Project Task

You are tasked with implementing a basic Unix-compatible Shell. The specification below is divided into parts that logically mirror the components required to develop a Unix-compatible Shell. If in doubt, test a specification rule/component against tcsh, the default shell available on linprog.cs.fsu.edu.

Conventions

- text – verbatim output
- (comment) – Explanation of a detail.
- [STRING] – Substitute this with an actual environment/shell state variable by that name.
 - Example: [USER]#> → cabrera#>
- {key combination} – Keyboard input.

Part 1: Command Line Parsing

Before the shell can begin executing commands, it must know what the commands to execute consist of. This includes the command name, the arguments, input/output re-direction, piping, and background execution indicators. In order to extract this information from the user input, you must parse the command line. Understand the following segments of the project prior to designing your parsing- the order of execution of many constructs may influence your parsing strategy. It is also critical that you understand how to parse arguments to a command and what delimits arguments.

A few details to be aware of:

- Beware leading whitespace:
 - # ls[NEW_LINE]
- Beware extraneous inter-command whitespace:
 - # ls | more |ls[NEW_LINE]
- Beware trailing whitespace:
 - # ls [NEW_LINE]

Part 2: Command Execution

Once the shell understands what commands to execute, now it is time to implement the execution of simple commands. The following rules describe the primary use cases:

- If a pathname begins with '/', use only the absolute path to test the existence of the target.
- If a pathname contains './', it shall be expanded to the current directory in the path resolution.
- A './' in the pathname requires searching only the parent directory.
- A '~/' at the start of a pathname indicates the pathname begins with \$HOME.
 - An arbitrary combination of the above symbols may occur in the path.
 - #~/../usr/local/../../bin/ls ~/cop4610t (only works if \$HOME is two directories deep)
- A '~' in any other portion of the pathname shall be treated as plain characters and shall not be expanded to \$HOME.
- If the pathname does not begin with any of the above, it requires searching each location in the \$PATH, in order, for the target.

To re-iterate, execution MUST use the `execv()` function. The remaining `exec*()` functions are disallowed. Four of the remaining `exec*()` functions perform path searching for you. The remaining `exec*()` functions is the most primitive, requiring you to construct the argument vector (`argv`) from a list of null-separated strings.

Part 3: Input/Output Re-Direction

Once the shell is capable of executing simple commands, we add a feature: the ability to re-direct output and input to and from files. The following set of rules describe the expected behavior:

- A command of the form 'CMD > FILE' shall be interpreted as CMD redirects its output to FILE.
 - If FILE does not exist, create FILE.
 - If FILE exists, overwrite FILE.
 - If CMD does not exist, treat FILE as above, but signal an error.
- A command of the form 'CMD < FILE' shall be interpreted as CMD receives through standard input the contents of FILE.
 - If FILE does not exist or is not a file, or if CMD does not exist or is not a command, signal an error.
 - If both CMD and FILE do not exist, the error should only signal that CMD does not exist.
- Commands of the form:
 - CMD <
 - < FILE
 - <
 - CMD >
 - > FILE
 - >
- ... shall signal an error.

Part 4: Pipes

Once your shell can re-direct input and output, it is capable of emulating the functionality of pipes. You now have a basis implemented with which to compare expected results. However, pipes are a required feature, so you must provide support for them. The following rules dictate how pipes should behave:

- A command of the form 'CMD₁ | CMD₂' shall be interpreted as CMD₁ redirects its standard output to CMD₂'s standard input.
- A command of the form 'CMD₁ | CMD₂ | CMD₃' shall execute in the order:
 - # CMD₁ | CMD₂
 - # [the result of CMD₁ | CMD₂] | CMD₃
- Commands of the forms:
 - |
 - CMD |
 - | CMD
- ... shall signal an error.

Part 5: Background Processing

With the majority of the primary user-interaction features implemented, you must now address the issue of running processes in the background. The rules for determining what commands should run in the background are as follows:

- A command of the form 'CMD &' should begin the execution of CMD in the background. Control shall return immediately to the shell.
 - When CMD begins execution, it shall print as follows:
 - [The position of CMD in the background execution queue] [CMD's PID]
 - When CMD completes execution and once the user presses enter:
 - [The position of CMD in the background execution queue]+ [CMD's command line]
- A command of the form '& CMD' executes CMD in the foreground. '&' is ignored.
- A command of the form '& CMD &' executes CMD in the background. The first '&' is ignored.
 - To summarize, a command shall run in the background if and only if its command line representation ends with '&'.
- A command of the form 'CMD₁ | CMD₂ &' shall run in the background.
 - When 'CMD₁ | CMD₂' begins execution, it shall print as follows:
 - [The position in the background execution queue] [CMD₁'s PID] [CMD₂'s PID]
 - When 'CMD₁ | CMD₂' completes execution and once the user presses enter:
 - [The position in the background execution queue]+ [CMD₁'s | CMD₂'s command line]
- A command of the form:
 - CMD₁ & | CMD₂ &
 - CMD₁ & | CMD₂
 - CMD₁ > & FILE
 - CMD₁ < & FILE
- ... shall signal an error.
- A command of the form 'CMD > FILE &' shall follow the above rules for output redirection and background processing, only control returns immediately to the shell.
- A command of the form 'CMD < FILE &' shall follow the above rules for input redirection and background processing, only control returns immediately to the shell.

Part 6: Built-ins

You are expected to support the following built-in commands:

- **exit** – terminates your running shell process and prints 'exit'.
 - [USER]:[PWD]# exit
 - exit
 - (shell exits)
- **cd** – Changes the present working directory.
 - **cd** with no arguments passed to it reverts the present working directory to \$HOME.
 - **cd** with one argument behaves as follows:
 - If the argument is '.', the PWD does not change.
 - If the argument is '..', the PWD changes to the parent.
 - If the argument is '~', the PWD changes to \$HOME.
 - If the argument is DIR, apply the path resolution rules to DIR:
 - If DIR is found and DIR is a directory, change the PWD to DIR.

- If not, signal that DIR does not exist.
 - If the argument is not a directory, signal an error.
- If there are more than one arguments, signal an error.
- **echo** – Outputs whatever the user specifies.
 - For each argument passed to echo:
 - If the argument does not begin with '\$':
 - Output that argument without modification to standard output.
 - If the argument begins with a '\$':
 - If the argument is an existing environment variable, expand the argument and output it.
 - Else, signal an error.
 - If any argument is in error, output the name of the first argument that is an error, and perform no further output.

Part 7: The Prompt

The prompt should always indicate to the user the PWD and who they are. Remember that `cd` can update the PWD. This is the format:

- [USER]:[PWD]#
- Example:
 - `cabrera:/home/cabrera#`

Part 8: A Near-Complete Shell

To verify the correctness of your shell, you should compare the execution of your shell against that of `tcsh`. A set of use cases has been provided. However, this set of use cases is not exhaustive, and your implementation will be tested against more.

Also, your shell should exit on two conditions:

- The command 'exit' is entered by the user.
- {Ctrl-D} is pressed by the user on an empty command line.
 - {Ctrl-D} sends EOF to the input.

Allowed Assumptions and Additional Comments

- You are allowed to assume that no more than three '|' will appear in a single command line.
- You do not need to handle special character expansion, e.g. command-line regular expressions.
 - `[USER]:[PWD]# ls /usr/include/*`
- You do need to handle the expansion of environment variables, as demonstrated above in the implementation of the **echo** built-in.
 - `[USER]:[PWD]# ls $HOME`
 - (outputs contents of home directory)
 - `[USER]:[PWD]# $SHELL`
 - (loads the standard system shell)
 - `[USER]:[PWD]# $UNDEFINED_FOO`
 - `UNDEFINED_FOO: Undefined variable`
 - `[USER]:[PWD]#`

- You can assume that the user input is no longer than 80 characters.
 - You do not need to test command lines consisting of more than 80 characters.
- You can assume pipes and input/output re-direction do not occur together.
- Single-level redirection: You can assume that if a '<' appears in a command line, another '<' will not appear.
 - The same applies for '>'.
- You do not need to implement auto-completion.
- You must be able to handle zombie processes.
- The above provided logical decomposition of the project tasks is only a suggested route. For example, you may choose to implement the built-ins after command line parsing, rather than last.
- You do not need to support any built-ins not specified (e.g. for, elif, ';').
- When testing against a tcsh shell, be sure you are running a tcsh shell.
- *NEW* You do not need to handle escaped spaces:
 - # ls some\ directory\ name
 - command: “/bin/ls”
 - args: “ls”, “some\”, “directory\”, “name\”

Project Submission Procedure

Follow the procedures given in the recitation syllabus. Remember to perform a 'make clean' prior to archiving your project. The webmail server discards all attachments containing an executable.