Project 1: Implementing a Shell

Due: TBA

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 create a new process, including search of the path, and an introduction to user-input parsing and verification. Furthermore, you will come to understand how input/output redirection, pipes, and background processes are implemented.

Problem Statement

Design and implement a basic shell interface that supports input/output redirection, 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). Unless otherwise specified, the required features should adhere to the operational semantics of the bash shell.

Project Tasks

You are tasked with implementing a basic shell. The specification below is divided into parts. When in doubt, test a specification rule against bash. You may access bash on linprog.cs.fsu.edu by logging in and typing the command bash. The default shell on linprog is tcsh. As specified in the project syllabus, 30 points of the project is based on documentation and 70 points are based on implementation. The distribution of the implementation points are listed next to each section.

Part 1: Parsing

Before the shell can begin executing commands, it needs to extract the command name, the arguments, input redirection (<), output redirection (>), piping (|), and background execution (&) indicators. Understand the following segments of the project prior to designing your parsing. The ordering 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. Code will be provided to help you understand the parsing process. You may use the provided code in your project but you are not required to. The provided code is merely an example to help you get started.

Part 2: Environmental Variables

Every program runs in its own environment. One example of an environmental variable is $USER, which expands to the current username. For example,

> echo $USER outputs:
rumancik

In the bash shell, you can type 'env' to see a list of all your environmental variables. You will need to use and expand various environmental variables in your shell, and you may use the getenv() library call to do so. For your built-in commands, you must expand environment variables used as arguments.

Part 3: Prompt [3]
The prompt should always indicate to the user the absolute working directory, who they are, and the machine name. Remember that cd can update the working directory. This is the format:

USER@MACHINE : PWD >

Example:

rumancik@linprog3 : /home/grads/rumancik/cop4610 >

Part 4: Shortcut Resolution [8]
You will need to convert different file path naming conventions to absolute path names. You can assume that directories are separated with a single forward slash (/).

- Shortcuts that can occur ***anywhere*** in the path
  - ..
    - Expands to the parent of the current working directory
    - Signal an error if used on root directory
  - .
    - Expands to the current working directory (the directory doesn't change)
- Shortcuts that can only occur ***at the start*** of the path
  - ~
    - Expands to $HOME directory
  - /
    - Root directory
- Relative paths
  - If the path does not start with “/”, the path is a relative path
  - Like a shortcut to the current working directory
  - Expands to the $PWD/PATH or $PWD/PATH
  - Signal an error if corresponding directory/file does not exist
- Files can only occur at the end of the path

Part 5: $PATH Resolution [7]
You will need to handle executable command paths slightly differently. If the path contains a '/', the path resolution is handled as above (Part 4), signaling an error if the end target does not exist or is not a file. Otherwise, if the *path* is just a single name, then you will need to prefix it with each location in the $PATH and search for file existence. The first file in the concatenated path list to exist is the path of the command. If none of the files exist, signal an error.

Part 6: Execution [8]
You will need to execute simple commands. First resolve the path as above. If no errors occur, you will need to fork out a child process and then use execv to execute the path within the child process. Commands need to support arguments (ie “ls -l /home/user/rumancik” in addition to “ls”).

Part 7: I/O Redirection [8]
Once the shell can handle simple execution, you'll need to add the ability to redirect input and output from and to files. The following rules describe the expected behavior, note there does not have to be whitespace between the command/file and the redirection symbol.

- CMD > FILE
  - CMD redirects its output to FILE
  - Create FILE if it does not exist
  - Overwrite FILE if it does exist
- CMD < FILE
  - CMD receives input from FILE
  - Signal an error if FILE does not exist or is not a file
- CMD < FILE_IN > FILE_OUT and CMD >FILE_OUT < FILE_IN
  - CMD receives input from FILE_IN
  - CMD outputs to FILE_OUT
  - Follows the rules specified above for individual redirection
- Signal an error for invalid syntax including the following:
  - CMD <
  - CMD >
  - < FILE
  - FILE >
  - <
  - >

Part 8: Pipes [8]
After your shell can handle redirection, you need to implement piping. Pipes should behave in the following manner (again, there does not have to be whitespace between the commands and the symbol):

- CMD1 | CMD2
  - CMD1 redirects its standard output to CMD2's standard input
- CMD1 | CMD2 | CMD3
  - CMD1 redirects its standard output to CMD2's standard input
  - CMD2 redirects its standard output to CMD3's standard input
- Signal an error for invalid syntax including the following
  - |
  - CMD |
  - | CMD

Part 9: Background Processing [8]

You will need to handle execution for background processes. There are several ways this can be encountered:

- CMD &
  - Execute CMD in the background
  - When execute starts, print
    [position of CMD in the execute queue] [CMD's PID]
  - When execution completes, print
    [position of CMD in the execution cue]+ [CMD's command line]
- & CMD
  - Executes CMD in the foreground
  - Ignores &
- & CMD &
  - Behaves the same as CMD &
  - Ignores first &
- CMD1 | CMD2 &
  - Execute CMD1 | CMD2 in the background
  - When execution starts, print
    [position in the background execution queue] [CMD1's PID] [CMD2's PID]
  - When execution completes, print
    [position in the background execution queue]+ [CMD1 | CMD2 command line]
- Must support redirection with background processing:
  - CMD > FILE &
  - CMD < FILE &
  - CMD < FILE_IN > FILE_OUT &
  - CMD > FILE_OUT < FILE_IN &
- Signal an error for invalid syntax including the following
  - CMD1 & | CMD2 &
  - CMD1 & | CMD2
  - CMD1 > & FILE
  - CMD1 < & FILE

Part 10: Built-ins [20]

Built-ins are part of the shell program itself. That is, they do not call execv to run an external executable command. The built-in commands you need to support are described below.

- exit [4]
  - If any background processes are still running, exit must wait for them to finish
  - Prints "Exiting..." along with the number of commands executed
  - Terminates the running shell process

- Example

  rumancik@linprog3 : /home/grads/rumancik/ > exit
  Exiting...
  Commands executed: 10
  (shell terminates)

- cd PATH [5]
  - Changes the present working directory according to the shortcut resolution above
  - If no arguments are supplied, it behaves as if $HOME is the argument
  - Signal an error if more than one argument is present
  - Signal an error if the target is not a directory

- echo ARGS [5]
  - Outputs whatever the user specifies
  - ***For each*** argument passed to echo
    - If the argument does not begin with “$”
      - Output the argument without modification
    - If the argument begins with “$”
      - Look up the argument in the list of environment variables
      - Print the value if it exists
      - Signal an error if it does not exist

- alias ALIAS_NAME=’CMD’ [4]
  - Allows user to store a name for a given command
  - CMD can be anything
    - external command or built-in command
    - may contain redirection, piping, background processing, etc
  - When user enters anything at command prompt, first check if entered string matches an ALIAS_NAME
    - if so, shell should behave as if CMD was entered as original input
  - Example

    > alias my_ls=’ls -lh > ls_out.txt’
    > my_ls
    writes output of ls -lh to ls_out.txt

- unalias ALIAS_NAME [2]
  - Remove ALIAS_NAME=CMD entry from list of aliases

Restrictions

- Must be implemented in the C Programming Language
- Only fork() and execv() can be used to spawn new processes
- You can not use system() or any of the other exec family of system calls
- Output must match bash unless specified above
- You may not use execv() in any of the built-ins (built-ins must be implemented “from scratch”)
- There is no limit on the number of characters per instruction, therefore, input must be handled dynamically

Allowed Assumptions

- Error messages do not need to match the exact wording of the bash shell, but should indicate the general cause of the error
- strtok() is permitted but strongly discouraged
- No more than two pipes (|) will appear in a single line
- You do not need to handle globs, regular expressions, special characters (other than the ones specified), quotes, escaped characters, etc
- You do not need to worry about expanding environment variables which are arguments to non-built-in commands
- There will be no more than 10 aliases
- Pipes and I/O redirection will not occur together
- Multiple redirections of the same type will not appear
- You do not need to implement auto-complete
- You do need to handle zombie processes
- The above decomposition of the project tasks is only a suggestion, you can implement the requirements in any order
- You do not need to support built in commands that are not specified

Extra Credit [up to 5 points]

- Support multiple pipes (limited by the OS and hardware instead of just 2), input redirection, and output redirection all within a single call [2].
- A novel, useful utility of your choosing. You must provide a written specification on the proper operation of your utility. Some examples include printing other proc files, simple auto-completion, if statements built-ins, loop built-ins, etc. Make sure your utility does not interfere with the other commands (including output) [2].
- Shell-ception: can execute your shell from within a running shell process repeatedly [1]
- *Extra credit activities must be documented in the README to receive credit*

Before submitting, please review the recitation syllabus for submission procedures.