Project 1: Implementing a Shell

Due: Friday February 9th, 2018, 11:59pm

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). 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.

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.

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, if my current username is 'dennis', typing:

```
=> echo $USER
```

outputs:

```
dennis
```

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. The getenv() procedure searchs the environment list for a string that matches the string pointed to by name. The strings are of the fom:

```
NAME = VALUE
```

Part 3: Prompt

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:

dennis@linprog3 :: /home/grads/dennis/cop4610t =>

Part 4: Path Resolution

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 (`/`).

- Directories 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)
  - `DIRNAME`
    - Expands to the child of the current working directory named `DIRNAME`
    - Signal an error if `DIRNAME` doesn't exist
    - Signal an error if `DIRNAME` occurs before the final item and is not a directory
- Directories that can only occur at the start of the path
  - `~`
    - Expands to `$HOME` directory
  - `/`
    - Root directory
- Files that can only occur at the end of the path
  - `FILENAME`
    - Expands to the child of the current working directory named `FILENAME`
    - Signal an error if `FILENAME` doesn't exist

You will need to handle commands slightly differently. If the path contains a '/', the path resolution is handled as above, 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 5: Execution

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.

Part 6: I/O Redirection

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
- Signal an error for the following
    - `CMD <`
    - `< FILE`
    - `<`
    - `CMD >`
    - `> FILE`
    - `>`

Part 7: Pipes

After it can handle redirection, your shell is capable of emulating the functionality of pipes. 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
- `CMD1 | CMD2 | CMD3 | CMD4`
    - `CMD1` redirects its standard output to `CMD2`'s standard input
    - `CMD2` redirects its standard output to `CMD3`'s standard input
    - `CMD3` redirects its standard output to `CMD4`'s standard input
- Signal an error for the following
    - `|`
    - `CMD |`
    - `| CMD`

Part 8: Background Processing

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

- `CMD &`
    - Execute `CMD` in the background
    - When execution starts, print
      [position of `CMD` in the execution queue] [`CMD's PID`]
    - When execution completes, print
      [position of `CMD` in the execution queue]+ [`CMD`'s command line]
- `& CMD`
    - Executes *CMD* in the foreground
    - Ignores &
- `& CMD &`
    - Behaves the sames 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]
- `CMD > FILE &`
  - Follow rules for output redirection and background processing
- `CMD < FILE &`
  - Follow rules for input redirection and background processing
- Signal an error for anything else
  - Examples includes
    - `CMD1 & | CMD2 &`
    - `CMD1 & | CMD2`
    - `CMD1 > & FILE`
    - `CMD1 < & FILE`

Part 9: Built-ins

- **exit**
  - Terminates your running shell process and prints “Exiting Shell...”
  - Example
    ```
    dennis@linprog3 :: /home/grads/dennis/cop4610t => exit
    Exiting Shell....
    ```
    (shell terminates)
- **cd `PATH`**
  - Changes the present working directory according to the path resolution above
  - If no arguments are supplied, it behaves as if `$HOME` is the argument
  - Signal an error if more than one argument is present
  - Signal an error if the target is not a directory
- **echo**
  - Outputs whatever the user specifies
  - For each argument passed to echo
    - If the argument does not begin with “$”
      - 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
- **etime `COMMAND`**
  - Record the start time using `gettimeofday()`
  - execute the rest of the arguments as per typical execution
    - You don’t have to worry about nesting with other built-ins
  - Record the end time using `gettimeofday()`
  - Output the elapsed time in the format of `s.us` where *s* is the number of seconds and *us* is the number of micro seconds (0 padded)

- Example

```
dennis@linprog3 :: /home/grads => etime sleep 1
Elapsed Time: 1.000000s
```

- **io COMMAND**
  - Execute the supplied commands
    - Again you don't have to worry about nesting with other built-ins
  - Record `/proc/<pid>/io` while it executes
  - When it finishes, output each of the recorded values
    - Your output comes after the command finishes
    - Your output needs to be in a table format (unlike the `io` file)
  - Example

```
dennis@linprog3 :: /home/grads => limits sleep 1
rchar:                  0
wchar:                  0
syscr:                  0
syscw:                  0
read_bytes:             0
write_bytes:            0
cancelled_write_bytes:  0
```

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
- You can not use any tokenizing functions like `strtok()`
- Output must match bash unless specified above

Allowed Assumptions

- No more than three 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
- Yo do need to handle expansion of environment variables
  - ls `$HOME`
    - prints contents of home directory
  - `$SHELL`
    - loads standard shell
  - `$UNDEFINED`
    - error, undefined environment variable
- The user input will be no more than 255 characters
- 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

Support multiple pipes (limited by the OS and hardware instead of just 3), input redirection, and output redirection all within a single call.

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).

Submission Procedure

Submit a tar archive of your code (no binaries or executables), Makefile, and README to the Canvas as is detailed in the recitation syllabus. Only submit once per team.