Project 1: Implementing a Shell

Due: February 8th, 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 $PATH environment variable, and an introduction to user-input parsing and verification. Furthermore, you will come to understand how input/output redirection, piping, and background processes are implemented.

Problem Statement

Design and implement a basic shell interface that supports input/output redirection, piping, 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 circumstances 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

When the user types in a command at your shell's prompt, your shell must parse the command string into meaningful tokens. The string may include a command name, some arguments, input redirection (<), output redirection (>), piping (|), and background execution (&) tokens. For the sake of this project, you may assume that the user enters commands where tokens are separated by a single space. Code will be provided to parse the command string into separate tokens, but you will need to build upon this code for further parsing.

Part 2: Environmental Variables [3]

Every program runs in its own environment. In the bash shell, you can type 'env' to see a list of all your environmental variables. Your job is to replace every token that starts with a dollar sign character into its corresponding value. For example, tokens ["echo", "$USER"] should expand to ["echo", "wkim"]. The output should look like

```
> echo $USER  wkim
```

You can do this using the getenv() function. On how to use getenv, read its man page. Note that this expansion happens no matter the command.

Part 3: Prompt [2]

For this project, the prompt should indicate the absolute working directory, the user name, and the machine name. You can do this by expanding the $USER, $MACHINE, $PWD environment variables. The format should be

```
        USER@MACHINE : PWD >
```

In my case, it looks something like this when I'm in my home directory on linprog:

```
        wkim@linprog3 : /home/grads/wkim >
```

Part 4: Tilde Expansion [5]

In Bash, tilde (~) may appear at the beginning of a path, and it expands to the environment variable $HOME. For example, tokens ["ls", "~/dir1"] should expand to ["ls", "/home/grads/wkim/dir1"]. An output example of tilde expansion is

```
> echo ~/aaa
/home/grads/wkim/aaa
```

Note that this expansion happens no matter the command, and that a standalone tilde should expand to $HOME. As a side note, tilde expansion in Bash is quite a bit more extensive. If interested, refer to https://www.gnu.org/software/bash/manual/html_node/Tilde-Expansion.html.

Part 5: $PATH Search [5]

When you enter "ls" in Bash, the shell somehow knows to execute the program located at "/usr/bin/ls". This also happens for commands like "cat", "head", "tail", and so on. This is not some sorcery, but rather a simple search of a predefined list of directories. This list of directories is defined in the environment variable called $PATH.

When the command does not include a slash (/) or is not a built-in function (part 10), you will need to search every directory specified in $PATH. Note that $PATH is actually just a long string where the directories are delimited by a colon. So you will need to perform some string operations.

If the command does not exist in any of the directories specified in $PATH, you will have to display an error message. In Bash, this is when you see the "command not found" error.

Part 6: External Command Execution [10]

Either the command included a slash or you had to carry out the $PATH search, you now have a path to the program you are trying to execute. Now, we only have to execute it. However, executing an external command turns out to be a little more work than a single line of the execv() function.

To put it simply, we have to fork() and execute the command in the child process using execv(). Note that you must handle commands with arguments correctly (e.g. ls -al).

Part 7: I/O Redirection [10]

Now that your shell can execute external commands, we are going to build up more functionalities around it in parts 7, 8, and 9. These functionalities only apply to external commands. In other words, you do not need to implement them for built-in functions (part 10).

In this section, we are going to implement I/O redirection from/to a file. Normally, the shell receives input from the keyboard and writes outputs to screen. Input redirection (<) replaces the keyboard and output redirection (>) replaces the screen with a specified file.

I/O redirection should behave in the following manner

- CMD > FILE
  - CMD writes its standard output to FILE
  - Create FILE if it does not exist
  - Overwrite FILE if it does exist
- CMD < FILE
  - CMD receives its standard 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 ○ Follows the rules specified above

Part 8: Piping [10]

The second functionality we are going to add to external command execution is piping. In a way, piping is an advanced form of I/O redirection. A difference to note is that piping requires more than 1 command. Instead of redirecting the I/O with a file, piping redirects the output of the first command with the input of the second command. For this project, there will only be a maximum of 2 pipes in any one command.

Piping should behave in the following manner

- 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

Part 9: Background Processing [10]

The last functionality is background processing. Up to this point, the shell waited to prompt for more user input when there were any external commands running. Background processing is a way to tell your shell to not wait for the external command to finish. Even though we told the shell to not wait for the command to finish, we are still interested in its completion. So your shell has to check up on the command's completion once in a while.

Note that background processing should also work along with I/O redirection or piping.
Background processing should behave in the following manner

- CMD &
    - Execute CMD in the background
    - When execute starts, print

      [Job number] [CMD's PID]
    - When execution completes, print

      [Job number]+ [CMD's command line]
- CMD1 | CMD2 &
    - Execute CMD1 | CMD2 in the background
    - When execution starts, print

      [Job number] [CMD2's PID]
    - When execution completes, print

      [Job number]+ [CMD1 | CMD2 command line]
- Must support redirection with background processing:
    - CMD > FILE &
    - CMD < FILE &
    - CMD < FILE_IN > FILE_OUT &
    - CMD > FILE_OUT < FILE_IN &

Part 10: Built-In Functions [15]

At this point, we are done with external command execution. Another aspect of the shell consists of internal commands that the shell natively supports. These are often called the built-in functions, and you will have to implement a few of them in your shell.

- exit [2]

- If any background processes are still running, you must wait for them to finish
- Print how long the shell has been running
- Print how long it took for the longest running command to execute
- Example:

```
> exit
Shell ran for 132 seconds and took 15 seconds to execute one command.
```

- cd PATH [5]
  - Changes the current working directory
  - If no arguments are supplied, change the current working directory to $HOME
  - Signal an error if more than one argument is present
  - Signal an error if the target is not a directory
  - Signal an error if the target does not exist
- echo ARGS [3]
  - Outputs whatever the user specifies
  - Example:

```
> echo ABC $USER abc  ABC
wkim abc
```

- jobs [5]
  - Outputs a list of active background processes
  - Example:

```
> jobs
[Job number]+ [CMD's PID] [CMD's command line]
```

Restrictions

- Must be implemented in the C Programming Language (No C++)
- 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
  - e.g. execl, execlp, execle, execvp, execvpe, etc
- Output must resemble Bash unless specified above
- You may not use execv() in any of the built-ins (must be implemented from scratch)
- Your Makefile should produce an executable file named *shell* when type *make* on your project root directory

Allowed Assumptions

- Error messages do not need to match the exact wording of Bash, but should indicate the general cause of the error

- No more than two pipes (|) will appear in a single command
- You do not need to handle globs, regular expressions, special characters (other than the ones specified), quotes, escaped characters, etc
- There will be no more than 10 background processes at the same time
- Piping and I/O redirection will not occur together in a single command
- Multiple redirections of the same type will not appear in a single command
- You do not need to implement auto-complete
- You only need to expand environment variables given as whole arguments
- The above decomposition of the project tasks is only a suggestion, you can implement the requirements in any order

Extra Credit [up to 5 points]

- Support unlimited number of pipes [2]
- Support piping and I/O redirection in a single command [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.