CS 377: Operating Systems Lab

Assignment 3

The purpose of this assignment is to understand UNIX processes, input/output streams of processes, and signals by constructing an interactive UNIX shell.

A UNIX shell is a simple textual interface to the operating system. Most shells expose the file system to the user who can execute various utilities and commands present in the file system to interact with the operating system. There are several popular UNIX shells: sh (the Bourne shell), csh (the C shell), zsh (the Z shell), and bash (the Bourne Again shell).

Your job in this assignment, which will span over two weeks, and three parts (we like trilogies), will be to create a program *Just Another Shell* (*JASH*), an interactive shell. This shell will expose the file system to the user and the user can run executable files present in the file system. The final version would support interactive interface, running batch files, parallel and sequential executions, background processes, input/output redirection, piping and cron.

PART A: A New Shell

23 January 2015

The following sections guide you through the first part of the assignment. **Please read the entire assignment and the coding and submission directives before you begin.**

The next lab will extend on this one. Therefore, it is important that this assignment be completed in order for you to expand it for the next lab assignment. Hence, even if you are unable to complete it in the given time frame, you must complete it before next lab to hope to do the next lab well.

You were told to read about certain functions and commands. If you have not done so, we recommend that you do so before you begin the assignment. You are expected to complete the assignment, as usual, in teams of two. And importantly, observe the deadline. **We will be strictly NOT accepting any late submissions.** Try not to keep the submission part till the last minute; Moodle is not really reliable for submitting your assignment 30 seconds before the deadline.

Interactive Operation

After startup, *jash* should perform the following actions in a loop:

1. Print a prompt consisting of a **$** sign followed by a space.
2. Read a line from standard input.
3. Lexically analyse the line to form an array of tokens.
4. Syntactically analyse (i.e. parse) the token to form a command.
5. Execute the command(s) or schedule their execution.

To begin with you are provided with a file named '***jash.c***' which does the first three parts for you (and partly the fourth part). You may use it as your starting point. If you wish to better model your code, or if you wish to use C++ (though it does not give added benefit), you may do so.

The program given breaks the input command into tokens. The program also takes care of inputs such as arguments given in quotes. Please go through the code and its documentation to understand what exactly it does.

Execution of Commands

A command can be either a file name which needs to be executed or a built-in command. The function ***execute_command(****…****)*** is intended for this purpose. It takes the tokens from the parser and based on the type of command proceeds to perform the necessary actions.

File Executions

Any command that is not a built-in command should be assumed to be a file that has to be executed. You have to search for the file name in the current directory and the directories mentioned in the **PATH** variable inherited from the parent shell. Look for ***execv*** variants that can do this. Display an error message if the file is not present in any of the directories. All the tokens following the file name are to be passed to the executable file as arguments.

Currently you are to not support background execution of processes (i.e. commands ending in ampersand `&'). All child processes forked by *jash* should run in the foreground. However, the user must be able to kill the current child process by sending it a SIGINT (Ctrl+C) signal. SIGINT should not kill *jash* itself.

Built-in Commands

1. **Change Directory (*cd*)**

Syntax: cd <absolute/relative path>

'cd' takes one argument which is the absolute or relative path of the directory you wish to switch to. Display an error message if the directory does not exist. (Read about ***chdir*** for more.)

2. **Batch File Execution (*run*)**

Syntax: run <file name>
Here <file name> gives the complete relative or absolute path to the file too.

'run' takes one argument which would be a batch file name containing a list of commands separated by newline. Your shell program should create a child process for the 'run' command instance and the this should then execute the commands one by one.

If one of them executes erroneously you should print the error and continue executing the subsequent commands. Once all commands are executed, the child process (created for the 'run' instance) without exiting the shell (parent process). Even if one or more of the commands within the batch file produces an error, the overall ‘run’ command will result in success. The overall ‘run’ command will return failure only if either creation of child process fails, file of the given name does not exists (or cannot be accessed) etc.

3. **Parallel Execution (*parallel*)**

Syntax: parallel <command 1> ::: <command 2> ::: … ::: <command *n*>

‘parallel’ executes commands concurrently, or in parallel. Each command is of the form of one these mentioned here. Each command in the given list of *n* commands ($n > 0$) is to be executed in parallel in separate child processes.
Example: parallel ls ::: pwd ::: echo abc
In this, three child processes must be created to execute each of the three commands.

4. **Sequential Execution (*sequential*)**

Syntax: sequential <command 1> ::: <command 2> ::: … ::: <command *n*>

‘sequential’ executes commands sequentially, and in a way of the ***&&*** operator in the bash or other shells. It is a lazy implementation of the boolean AND operator.
More precisely, it executes the commands sequentially until any command produces an error, in which case it does not proceed to the next command.

Example 1: sequential ls ::: pwd ::: echo abc
In this, each of the three processes will be executed sequentially.
Example 2: sequential ls ::: invalid ::: echo abc
In this case, since ‘invalid’ is not a valid command, the command(s) after ‘invalid’ (in this case, ‘echo abc’) will not be executed.

5. **Exit**

‘exit’ command exits the *jash* program immediately. *jash* should also exit on receiving the EOF (or Ctrl+D) character on input. (Just like your bash does.). Other than this, you should ensure that other signals such as SIGINT are caught appropriately. If any child process is running in the foreground Ctrl+C or Ctrl+\ should kill it, but should not kill the *jash* program (parent process). You will have to design appropriate signal handlers for SIGINT (Ctrl+C), SIGQUIT (Ctrl+\) and any other signals you are using.

6. **File Executions and Error**

Any command not of the other forms is a file execution. If the file does not exist the command is to be treated as invalid. In such a case, an error is to be printed and the whole input line is to be discarded. Print all error messages to STDERR stream only.

7. **(BONUS) Sequential Execution (*sequential_or*)**

Syntax: sequential_or <command 1> ::: <command 2> ::: … ::: <command *n*>

'sequential_or' executes commands sequentially, and in a way of the || operator in the bash or other shells. It is a lazy implementation of the boolean OR operator.
More precisely, it executes the commands sequentially until any command does not produce an error, in which case it does not proceed to the next command. As soon as a command executes successfully, further processes are not executed.

Example 1: sequential ls ::: echo abc
In this case, if 'ls' executes successfully (it most likely will) 'echo abc' will not be executed.
Example 2: sequential invalid ::: echo abc
In this case, since 'invalid' is not a valid command, 'echo abc' will be executed.

8. **(BONUS) Combined Commands**

Combined commands means two or more built-in commands put together. You will get additional credit if your program can handle such cases.

Example 1: parallel ls ::: run run_file.txt ::: pwd
Example 2: sequential cd test ::: pwd ::: ls
Example 3: "run" or "parallel" or "sequential" commands within a batch file provided to "run"

Attempt bonus sections only if you've done previous parts. Don't attempt for 6 pm deadline.

Tips and Recommendations

1. We recommend you follow the structure mentioned in the code to write *jash*. Each section should be implemented as a function, which may call on any number of helper functions you define. You can overload a function for a section if you want to pass a different number of arguments for different cases.
2. Use a variant of the ***execv*** command to supply arguments and search for executable in the directories mentioned in the PATH variable. See the man page of ***execv*** for details. Use the ***chdir*** system call to implement the 'cd' command. You may use the ***waitpid*** call to wait for a child process to complete execution or define a handler for the SIGCHLD signal.
3. After printing anything to STDOUT, flush the output. Use ***fflush (stdout)***.
4. Read the error codes returned by various functions carefully, especially ***exec*** or ***execv*** and its variants, ***malloc***, ***fopen*** etc. and use them to handle and display any errors. All errors must be handled by your program, and graciously displayed.
5. Use man pages for description of system calls. You can find consolidated data for most of the required commands here also: http://pubs.opengroup.org/onlinepubs/007908799/xsh/unistd.h.html
6. If using the given code and building upon it, for 'run', 'sequential' and 'parallel' try to make (recursive) calls to ***execute_command(…)*** function. That way you wouldn't have to specially handle combined commands.
7. Nested parallel command, nested sequential command or combination of parallel and sequential in a single command will not be given as a test case (even for the bonus sections); you need not worry about parsing such cases.

Coding Guidlines and Directives

1. You may code in C or C++ only. You may have multiple source and/or header files. You must adhere to the following **directory structure**:
   <RollNumber1>_<RollNumber2>_lab3a
   - [*all source/header files*]
   - Makefile
   - README.txt
   - **test** (optional)

2. **README.txt** file must contain your details (names and roll numbers) , implementation status, additional documentation and anything else you need to tell us. You may optionally add a "**test**" folder containing test cases which worked for your program. This will help us grade.

3. **Makefile** is given to you. It is designed to handle code from multiple source and header files. You may use the given file, or modify it as needed. But a Makefile is needed. Moreover it should have the targets "***all***" (which compiles all source files into the target called '*jash*') and "***clean***" (which clears all temporary, object and executable files).

4. *jash* should handle an **error cases** gracefully by rejecting the line and writing a descriptive error message to the STDERR stream. Try to use the ***errno*** variable ***perror*** function for this. The *jash* program must NOT exit unnecessarily or result in a Segmentation Fault.

5. Your program should contain no **memory leaks**. For every call of ***malloc*** (or ***new***), eventually there should be a call of ***free*** (or ***delete***).

6. Assume that no input line, either from STDIN or a batch file, would contain more than 1000 characters, the newline character included.

7. **Documentation is NECESSARY**. Sorry to bother you, but it is. Each function definition should have a comment stub above it to describe what it does. As a healthy programming practice, you should have ample number of comments inside function definitions to guide someone who is reading your code for the first time.

Note that failure to adhere to these directives will result in loss of marks.

Submission Instructions

1. Create a single tar-zipped archive of your folder (which you have to submit) —
tar zcvf <RollNumber1>_<RollNumber2>_lab3a.tgz <RollNumber1>_<RollNumber2>_lab3a

2. There are **two submission deadlines**. Separate links will be put up on Moodle for both submissions. For the early (**6 pm**) deadline you have to submit everything except the "parallel" and "sequential" commands. For the final deadline (**10 pm**) you will have to submit everything. Submission for the early deadline is compulsory. Even if your assignment is incomplete, please submit it for the early (6 pm) deadline