EC440: Project 1- Shell

Project Goals

- To understand/implement important Unix system calls.
- To develop a simple shell application.

Collaboration policy

- You are encouraged to discuss this project with your classmates/instructors but are required to turn in your own solution.
- You must be able to fully explain your solution during oral examination.

Deadlines

- Sept. 24th 23:59 EDT; more details on process will be provided in updates of this project description. You should get started ASAP; this is a very challenging project, and check piazza resources regularly for updates.

Project Description

The goal of this project is to implement a basic shell written in C and running on Linux. It should be able to execute commands, redirect the standard input/output of commands to files, pipe the output of commands to other commands, and carry out commands in the background.

During startup, your program must print "my_shell" (without quotes). After this, the user should be able to type commands (e.g. **`ls`**,**`ps`**,**`cat`**) to be executed. You can access these binaries by searching directories determined by the PATH environment variable that is passed to your shell. Commands can have arguments that are separated by whitespace. For example, if the user types **`cat x`**, your shell will need to invoke **`cat`** binary and pass **`x`** as its argument. When the shell has received a line of input, it typically waits until all commands have finished. Only then, a new prompt is displayed (however, this behavior can be altered - see below for details).

Your shell must also be able to interpret and execute the following meta-characters: '<', '>', '|', and '&':

- **`command < filename`** : In this case, a command takes its input from the file (not stdin). Note that spacing is irrelevant in this case. For example, **`cat<file`** and **`cat <file`** are valid inputs. Also, only one input redirection is allowed for a single command. (**`cat <<file`** is invalid)
- **`command filename`** : An input following this template indicates that a command writes its output to the specified file (not stdout). Again, spacing is irrelevant and only one input redirection is allowed for a single command.
- **|** pipe sign allows several commands to be connected: the output of the command before "|" is piped to the input of the command following "|". Multiple pipe signs are allowed on the command line. Spacing is irrelevant (described above). Example: **`"cat a | sort | wc"`** `(without quotes)` indicates that the output of the **`cat`** command is channeled to the **`sort`** and **`sort`** sends its output to the input of the **`wc`** program.
- The ampersand character '**&**' should allow user to execute a command(s) in the background.

In this case, the shell immediately displays a prompt for the next line regardless of whether the commands on the previous line have finished).

For simplification purposes, you should assume that only one '**&**' character is allowed and can only appear in the end of the line. Also, if input line consists of multiple commands, only the first command on the input line can have its input redirected, and only the last can have its output redirected. In case of a single command, standard rules apply (i.e., **`cat < x > y`** is valid, while **`cat f | cat < g`** is not).

The parser program you create should be called **`myparser`** and should interpret a command line as described above. Its output should consist of tokens, one per line, with a description of their interpretation (eg, metacharacter, command, command argument). TA's will post some examples.

In case of errors (if the input does not follow the rules/assumptions described above, command is not found, etc.), your shell should display an error message (cannot exceed a single line) and wait for the next input. "ERROR:" (without quotes) + *your_error_message*.

~~To facilitate automated grading, when you start your simple shell program with the argument 'n', then your shell must not output any command prompt (no "shell: "). Just read commands as usual~~.

To exit the shell, the user must type Ctrl-D (pressing the D button while holding control).
You may assume that the maximum length of individual tokens (commands and filenames) is 32 characters, and that the maximum length of an input line is 512 characters.

Your shell is supposed to collect the exit codes of all processes that it spawns. That is, you are not allowed to leave zombie processes around of commands that you start.

Your shell should use the *fork(2)* system call and the *execvp(2)* system call (or one of its variants) to execute commands. It should also use *waitpid(2)* or *wait(2)* to wait for a program to complete execution (unless the program is in the background). You might also find the documentation for signals (and in particular SIGCHLD) useful to be able to collect the status of processes that exit when running in the background.

Discussion and hints:

1. A simple shell such as this needs a command-line parser to figure out what the user is trying to do. To read a line from the user, you may use *fgets(3)*. The parser should be able to read an input line and identify the special characters, command (s) and other tokens.
2. If a valid command has been entered, the shell should *fork* to create a new (child) process, and the child process should *exec* the command.
3. Before calling *exec* to begin execution, the child process may have to close stdin (file descriptor 0) or stdout (file descriptor 1), open the corresponding file or pipe (with open for files, and pipe for pipes), and use *dup2(2)* or *dup* to make it the appropriate file descriptor. After calling *dup2*, close the old file descriptor.
4. The main challenge of calling *execvp* is to build the argument list correctly. If you use *execvp*, remember that the first argument in the array is the name of the command itself,

and the last argument must be a null pointer.

5. The easiest way to redirect input and output is to follow these steps in order: (a) open (or create) the input or output file (or pipe). (b) close the corresponding standard filedescriptor (stdin or stdout). (c) use *dup2* to make file descriptor 0 or 1 correspond to your newly opened file. (d) close the newly opened file (without closing the standard file descriptor).
6. When executing a command line that requires a pipe, the pipe must be created before forking the child processes. Also, if there are multiple pipes, the command(s) in the middle may have both input and output redirected to pipes. Finally, be sure the pipe is closed in the parent process, so that termination of the process writing to the pipe will automatically close the pipe and send an EOF (end of file) to the process reading the pipe.
7. Any pipe or file opened in the parent process may be closed as soon as the child is forked - this will not affect the open file descriptor in the child.

Submission Guidelines

Ultimately you will upload your suite of code (makefile, C source) to be graded along with your oral examinations/demonstration. We are working on setting up the servers appropriately and will give you detailed instructions before the due date.