Project 3

`1730sh` - A Unix Shell

Pair Programming Project

DUE TUE 2015-12-08 @ 11:55 PM

CSCI 1730 – Fall 2015

Project Description

For this project, you and your partner are going to implement a Unix shell (similar to bash) in C++ that supports background/foreground job control, pipelining, and IO redirection. Your code will make use of low-level system calls such as `fork`, `exec`, `pipe`, and related functions. You are **NOT** allowed to use the `system`, `popen`, and `pclose` system calls or functions.

Functional Requirements

Here you will find detailed information about the functional requirements for your shell.

- **Prompt (20 points):**

  Your shell should read input from the prompt and divide it into words and operators, employing quoting rules (see below) to define the meaning of each character of input. Then these words and operators are translated into commands and other constructs, which return an exit status available for inspection or processing.

  ```
  1730sh:~/cs1730/p3/$ cat filename
  ```

  Using double quotes, the literal value of all characters enclosed is preserved, except for the backslash character. The backslash retains its meaning only when followed by another backslash or newline. Within double quotes, backslashes are removed from the input stream when followed by a double quote character. In other words, a double quote may be quoted within double quotes by preceding it with a backslash. The following example would have `argc == 2`:

  ```
  1730sh:~/cs1730/p3/$ echo "my \"awesome\" shell"
  ```

  Your shell's prompt should include the text `"1730sh:"` followed by the present working directory, the `$` symbol, and a single space. If the present working directory is under the user's home directory, then a tilde (∼) should replace the absolute path of the home directory in the prompt.

  Note: Your shell should ignore the job control signals, however jobs executed using your shell should not. For example, typing `C-c` to send `SIGINT` should not terminate your shell.

- **Job Control (20 points):**

  In addition to job control facilitates that you have via your `kill` implementation (see Project 2), your shell needs to support launching jobs in both the foreground and the background. Your shell also needs to facilitate job bookkeeping.

  – *Bookkeeping:* Each job needs to have an associated job identifier (JID). I recommend placing all the processes associated with a job into the same process group, then using the process group id (PGID) as the JID. When a job encounters a status change (e.g., termination, stopped, continued, etc.), your shell needs to display a message letting the user know about the status change. This output should be similar to the output of the `jobs` builtin command (detailed later in this document). Here are some examples of status change messages:

  ```
  1137 Stopped           less &
  2245 Continued         cat /dev/urandom | less &
  2343 Exited (0)        emacs
  2345 Exited (SIGKILL)  emacs
  ```

- *Foreground Jobs:* Nothing special needs to be done in order to launch a job in the foreground. When a foreground job is launched, your shell must wait until the job encounters a status change before re-prompting the user.

- *Background Jobs:* A job may be launched in the background by appending an ampersand (&) after the job in your prompt. When a job is run in the background, your shell must **NOT** wait before re-prompting the user.

- **Pipelining (20 points):**

  Your shell needs to support pipelined jobs, where the standard outputs and standard inputs of the processes in the job are chained together using a Unix pipe (see `pipe(2)`). Remember, you are not allowed to use `popen` or `pclose`. A user should be able to execute a pipelined job using your shell by separating the individual commands by the vertical bar (|) operator. For example:

  ```
  1730sh:~/cs1730/p3/$ cat file | grep // | less
  ```

  In the example above, there are two pipes. The first pipe chains the standard out of the first process to the standard input of the second process, and the second pipe chains the standard out of the second process to the standard input of the third process.

- **IO Redirection (20 points):**

  Your shell needs to support the redirection of standard input, standard output, and standard error. In the examples below `COMMAND` may be a simple job or a pipelined job.

  - Redirect Standard Input using `<`

    ```
    1730sh:~/cs1730/p3/$ COMMAND < myfile.txt
    ```

  - Redirect Standard Output using `>` (truncate) or `>>` (append)

    ```
    1730sh:~/cs1730/p3/$ COMMAND > myfile.txt
    1730sh:~/cs1730/p3/$ COMMAND >> myfile.txt
    ```

  - Redirect Standard Error using `e>` (truncate) or `e>>` (append)

    ```
    1730sh:~/cs1730/p3/$ COMMAND e> myfile.txt
    1730sh:~/cs1730/p3/$ COMMAND e>> myfile.txt
    ```

  Your shell should support multiple redirections at once. For example, the following is considered to be valid:

  ```
  1730sh:~/cs1730/p3/$ COMMAND < input.txt > output.txt e>> log.txt
  ```

  For the purposes of this project, you may assume the following order for redirects will be respected: `<`, `>`, `>>`, `e>`, `e>>`. You may also assume that the truncate and append redirects for a particular file are mutually exclusive.

- **Additional Builtins (20 points):**

  Your shell needs to support the following builtin commands:

  - `bg JID` – Resume the stopped job JID in the background, as if it had been started with `&`.
  - `cd [PATH]` – Change the current directory to `PATH`. The environmental variable `HOME` is the default `PATH`.
  - `exit [N]` – Cause the shell to exit with a status of `N`. If `N` is omitted, the exit status is that of the last job executed.
  - `export NAME[=WORD]` – `NAME` is automatically included in the environment of subsequently executed jobs.
  - `fg JID` – Resume job JID in the foreground, and make it the current job.
  - `help` – Display helpful information about builtin commands.
  - `jobs` – List current jobs. Here is an example of the desired output:

    ```
    JID  STATUS       COMMAND
    1137 Stopped      less &
    2245 Running      cat /dev/urandom | less &
    2343 Running      ./jobcontrol &
    ```

Nonfunctional Requirements

Here you will find detailed information about the nonfunctional requirements for your shell.

- **User Input:** You may **NOT** assume valid user input except where noted.
- **Error Handling:** If a system call results in an error, then your implementation should display the error using `perror` (available in `<cstdio>`). In general, if an error occurs (e.g., invalid input), then display a message to the user indicating what the error is. Since this is a shell, you should not exit when an error occurs in the parent process. If an error occurs in a child process before an `exec`, then you should exit with status `EXIT_FAILURE` (available in `<cstdlib>`).
- **C++ Code & Structure:** Make sure that all of your files are in a directory called `LastName-FirstName-p3`, where `LastName` and `FirstName` are replaced with you and your partner's actual last names, respectively.
- `Makefile` **File:** You need to include a `Makefile`. Your `Makefile` needs to compile and link separately. Make sure that your `.cpp` files compile to individual `.o` files. The resulting executable for your shell should be `1730sh`.
- `README` File Make sure to include a `README` file that includes you and your partner's names and ID numbers as well as instructions on how to compile and run your program. *Note:* Try to make sure that each line in your `README` file does not exceed 80 characters. Do not assume line-wrapping. Please manually insert a line break if a line exceeds 80 characters.
- **Compiler Warnings:** Since you should be compiling with both the `-Wall` and `pedantic-error` options, your code is expected to compile without `g++` issuing any warnings. For this project, compiling without warnings will be one or more of the test cases.
- **Memory Leaks:** You are expected to ensure that your implementation does not result in any memory leaks. We will test for memory leaks using the `valgrind` utility. For this project, having no memory leaks will be one or more of the test cases.

Collaboration Policy

Students are required to work in groups of two for this project. Furthermore, each group needs to engage in pair programming. Pair programming is an agile software development technique in which two programmers work together at one workstation. One, the driver, writes code while the other, the navigator, reviews each line of code as it is typed in. The two programmers switch roles frequently.

Yes, this involves physically meeting with your partner. Not being able to find time is **NOT** an excuse for not participating. If you need a place to meet and work on the project then I suggest the 307 lab in Boyd. The iMacs are already setup with everything you need. You login to them with your Nike account, open up the "Terminal" application and SSH into your Nike account.

Some Pair Programming Guidelines

- You and your partner should work together as much as possible, with the stipulation that at most 25% of your total time coding, testing, and debugging on the assignment can be done alone.
- When the pair gets back together after either partner has worked on the code alone, review, line by line, the work done alone before doing any new work. This is really easy if each person maintains their own branch and commits as they work.
- You and your partner should alternate driving and navigating, spending roughly equal amounts of time in each role.
- After the project is completed, a pair programming survey will be made available for both partners to complete.

GitHub Repositories

Private Git repositories will be created on GitHub for each team. Information about these repositories will be posted on Piazza.

Fork Bomb Policy

Fork bombing Nike will hurt your grade. If I find out that you have fork bombed Nike, there will be a 25-point deduction from your project for the first offense. Log in to one of the `cf` cluster nodes to avoid this. Also, if you are forking processes within a loop, use a simple counter to make sure the loop doesnt execute more than a handful of times while you are still debugging. It is better to be safe than sorry.

Submission

Before the due date, you need to submit your code via Nike. Make sure your work is on `nike.cs.uga.edu` in a directory called `LastName1-LastName2-p3`. From within the parent directory, execute the following command:

```
$ submit LastName1-LastName2-p3 cs1730a
```

It is also a good idea to email a copy to yourself. To do this, simply execute the following command, replacing the email address with your email address:

```
$ tar zcvf LastName1-LastName2-p3.tar.gz LastName-FirstName-p3
$ mutt -s "p3" -a LastName1-LastName2-p3.tar.gz -- your@email.com < /dev/null
```