Due Friday, 2014-08-01 @ 11:55 PM

`simplush` is a shell that support job control that you will implement in your choice of either C or C++. The name is an acronym for 'SIMPLe Unix SHell' (you can come up with a cooler name for your implementation when the semester is over). At its base, your shell is simply a process that creates other processes in order to execute commands. It provides a user interface to the programs available on the system as well as built-in commands (builtins) implementing functionality impossible or inconvenient to obtain via separate utilities.

Initializing a Shell

Since your shell will support job control, it should set itself to ignore all the job control stop signals so that it doesn't accidentally stop itself. You can do this by setting the action for all the stop signals to `SIG_IGN` using `signal`. However this will result in any child processes created by your shell also ignoring these signals by inheritance. This is definitely undesirable, so each child process should explicitly set the actions for these signals back to `SIG_DFL` just after it is forked.

Shell Commands

Your shell will support builtin commands as well as commands for executing other programs. Here are some descriptions to help you get started.

Simple Commands

A simple command is the kind of command encountered most often. It's just a sequence of words separated by blanks, terminated by one of the shell's control operators. The first word generally specifies a command to be executed, with the rest of the words being that command's arguments. The return status of a simple command is its exit status as provided by the `waitpid` function, or `128+n` if the command was terminated by signal `n`. In general, you want to support quoted command-line arguments, but for this project you don't have to.

Builtin Commands

Your shell should implement the following builtin commands.

- **`cd [dir]`:** Change the current working directory to `dir`.
- **`exit [n]`:** Exit the shell, returning a status of n to the shell's parent. If n is omitted, the exit status is that of the last command executed.
- **`continue n`:** Send `SIGCONT` to job with job number n. This brings the job into the foreground and does any cleanup that might be necessary so that your shell knows the job was continued.
- **`delenv name`:** Remove any instance of an environmental variable called `name` in the environment list that is to be passed to child processes.
- **`jobs`:** Lists the jobs running in the background in the order in which they were executed, giving the *job number*, and incrementing integer value. The oldest job in the list always have a job number of 1. The output of this command should look something like the following:

```
[1] RUNNING  $ ls -a &
[2] STOPPED  $ ls -alh &
[3] STOPPED  $ ls -lh > lsout.txt &
```

The first value is the job's number. The second value indicates the current state of the job's process. The rest of the line show the command the user typed into the shell to execute the job.

- **`kill n :`** Sends `SIGTERM` to job with job number `n`. This also does any cleanup that might be necessary so that your shell knows the job was killed.

- **`pwd :`** Prints out the absolute path of the current working directory.

- **`setenv [name[=value]] :`** Set a `name` to be passed to child processes in the environment list. If no names are supplied, the current state of the environment list is displayed. By default the initial `name=value` strings in the environment list should be inherited from your shell's parent.

- **`thriller :`** Prints out the lyrics to Michael Jackson's 1982 hit, "Thriller."

Facilitating Redirection

Before a simple command is executed, its input and output may be redirected according to one or more redirection operators that may follow the command. Redirections should be processed in the order they appear, from left to right.

- **'`< filename`'** should redirect standard input to `filename`.

- **'`> filename`'** should redirect standard output to `filename`. If the file does not exist, it should be created with mode 0644. If the file does exist, it should be truncated upon opening.

- **'`>> filename`'** should redirect standard output to `filename` for appending. If the file does not exist, it should be created with mode 0644.

- **'`e> filename`'** should redirect standard error to `filename`. If the file does not exist, it should be created with mode 0644. If the file does exist, it should be truncated upon opening.

- **'`e>> filename`'** should redirect standard output to `filename` for appending. If the file does not exist, it should be created with mode 0644.

A failure to open or create a file should simply cause the redirection to fail. Your shell should be able to redirect standard output and standard error to the same file of whether or not the redirects are of the same type (i.e., truncating or appending).

Foreground and Background Jobs

If the last token in your command is not a '&', then you're going to execute a job for that command in the foreground. When a foreground job is launched, your shell must first give it access to the controlling terminal by calling tcsetpgrp (An easy way to know if you did this correctly is to try and run a terminal-heavy application like `emacs`). Then, your shell should wait for processes in that process group to terminate or stop.
When all of the processes in the group have either completed or stopped, your shell should regain control of the terminal for its own process group by calling `tcsetpgrp` again. Since stop signals caused by I/O from a

background process or a `SUSP` character typed by the user are sent to the process group, normally all the processes in the job stop together. If your shell executes a job in the background (command ends with '&'), it should remain in the foreground itself and continue to read commands from the standard input. Your shell will need to keep track of all its background jobs so that, among other things, you can implement the `jobs`, `continue`, and `kill` builtin commands.

Extra Credit XOR Group Option

Right now, the specification for your shell doesn't require you to handle quoted command-line arguments. If a command line argument is quoted, the quoted string is treated as a single argument to the to-be-executing program, regardless of whether or not there are blank spaces within the quoted string. Normal escaping applies. For example, a '\"' character does not start or end a quoted string since it's escaped.

- If you're working in a group, then you are required to implement this feature.
- If you're not working in a group, then you can earn up to 5 extra credit points for implementing this feature.

Extra Credit for Everyone

For a whopping 10 extra credit points, implement piping in your shell. UNIX philosophy urges the use of small yet highly focused programs that they can be used together to perform complex tasks. To do this, you need to direct the standard output of one program into the standard input of another program, but without the normal I/O redirection you're used to via `dup2`. Instead, multiple simple commands that are separated by the pipe operator, '|' are connected together by pipes so that they run together dynamically as data flows between them.

Makefile Requirements

Your Makefile should be setup so that there is a target that ensures each source file is individually compiled into a corresponding object file, an "all" target (placed as the first target) that depends on the object files and links them together to generate the "`simplush`" executable, and a "clean" target that removes the target executable and any object files that were generated during the build process.

Submission Instructions

Please do not submit your project with the executable and object files included. Only the source files, header files, and your Makefile should be included. Assuming you wrote your `Makefile` correctly, just run a "make clean" before you proceed to submit your project. Assuming all of your project files are in a directory called `p3` on Nike, change into the parent directory of `p3` and execute the following command to submit your project:

```
$ submit p3 cs1730
```