Assignment 3 - Simple Shell

Overview

In this assignment you will write your own shell in C. The shell will run command line instructions and return the results similar to other shells you have used, but without many of their fancier features.

In this assignment you will write your own shell, called simplesh. This will work like the bash shell you are used to using, prompting for a command line and running commands, but it will not have many of the special features of the bash shell.

Your shell will allow for the redirection of standard input and standard output, and it will support both foreground and background processes.

Your shell will support three built in commands: exit, cd, and status. It will also support comments, which are lines beginning with the # character.

Specifications

All execution, compiling, and testing of this program should be done from the bash prompt on the Cal Poly servers.

Use the colon : symbol as a prompt for each command line. Be sure you flush out the prompt each time you print it, this makes the test script look nicer.

The general syntax of a command line is:

```
command [arg1 arg2 ...] [< input_file] [> output_file] [&]
```

…where items in square brackets are optional. You can assume that a command is made up of words separated by spaces. The special symbols <, >, and & are recognized, but they must be surrounded by spaces like other words. If the command is to be executed in the background, the last word must be &. If standard input or output is to be redirected, the > or < words followed by a filename word must appear after all the arguments. Input redirection can appear before or after output redirection.

Your shell does not need to support any quoting; so, arguments with spaces inside them are not possible.

Your shell should support command lines with a maximum length of 2048 characters, and a maximum of 512 arguments. You do not need to do any error checking on the syntax of the command line.

Command Execution

You will use fork, exec, and waitpid to execute commands. The shell will wait for completion of foreground commands (commands without the &) before prompting for the next command. The shell will not wait for background commands to complete.

Background commands should have their standard input redirected from /dev/null if the user did not specify some other file to take standard input from. What happens to background commands that read from standard input if you forget this?

Your shell should use the PATH variable to look for commands, and it should allow shell scripts to be executed (see below for the testing script, for example). The right version of the exec function will do this for you automatically. If a command fails because the shell could not find the command to run, then the shell will print an error message and set the exit status to 1.

After the fork but before the exec you must do any input/output redirection. A redirected input file should be opened for reading only; if your shell cannot open the file for reading it should print an error message and set the exit status to 1. A redirected output file should be opened for write only, it should be truncated if it already exists or created if it does not exist. If your shell cannot open the output file it should print an error message and set the exit status to 1.

The built-in command status can be used to print the exit status of the last foreground command. If a command (either a foreground or background command) is terminated by a signal, a message indicating which signal terminated the process will be printed. The shell will print the process id of a background process when it begins. When a background process terminates, a message showing the process id and exit status will be printed. You should check to see if any background processes completed just before you prompt for a new command and print the message *then*. In this way the messages about completed background processes will not appear during other running commands, though the user will have to wait until they complete some other command to see these messages (this is the way the C shell and Bourne shells work). You will need to use waitpid to check for completed background processes.

Be sure that a CTRL-C interrupt from the keyboard does not terminate your shell, but only the foreground command it is running.

Background commands should not be terminated by a CTRL-C signal.

Built-in Commands

Your shell will support three built in commands: exit, cd, and status.

The exit command exits the shell. It takes no arguments.

The cd command changes directories. By itself, it changes to the directory specified in the HOME environment variable. It can also take one argument, the path of the directory to change to. Note that this is a working directory: when simplesh exits, the pwd will be the original pwd when simplesh was launched.

The status command prints out the exit status or terminating signal of the last foreground process. You do not have to support input/output redirection for these built in commands and they do not have to set any exit status.

Finally, your shell should allow blank lines and comments. Any line that begins with the # character is a comment line and should be ignored. A blank line (one without any commands) should do nothing; your shell should just reprompt for another command.

Example

Here is an example:

```
% simplesh
: ls
junk   simplesh    simplesh.c
: ls > junk
 : status
 exit value 0
: cat junk
junk
simplesh
simplesh.c
: wc < junk
       3       3      21
: test -f badfile
: status
exit value 1
: wc < badfile
simplesh: cannot open badfile for input
: status
exit value 1
: badfile
badfile: no such file or directory
: sleep 5
^Cterminated by signal 2
: status
terminated by signal 2
: sleep 15 &
background pid is 4923
: ps
   PID TTY      TIME CMD
  4923 pts/4    0:00 sleep
  4564 pts/4    0:03 tcsh-6.0
  4867 pts/4    1:32 simplesh
:
:
```

```
: # that was a blank command line, this is a comment line
background pid 4923 is done: exit value 0
: # the background sleep finally finished
: sleep 30 &
background pid is 4941
: kill -15 4941
background pid 4941 is done: terminated by signal 15
: pwd
/nfs/stak/faculty/b/brewsteb/CS344/prog3
: cd
: pwd
/nfs/stak/faculty/b/brewsteb
: cd CS344
: pwd
/nfs/stak/faculty/b/brewsteb/CS344
: exit
%
```

In addition to your shell needing to replicate the above example in functionality, this assignment is provided with test script. To run it, place it in the same directory as your compiled shell, and run this command from a bash prompt:

```
 % testscript 2>&1
```

or

```
% testscript 2>&1 | more
```

or

```
% testscript > mytestresults 2>&1
```

Don't worry if the spacing, indentation, or look of the output of the script is different than when you run it interactively: that won't affect your grade. The script may add extra colons at the beginning of lines or do other weird things. Use it to prepare for your grade but base the look and feel of your program on the interac ve running of your shell.

What to submit

Please submit a single zip file of your program code, which may be in as many different files as you want. Also, inside that zip file, provide a file called readme.txt that contains instructions on HOW to compile your code; you may compile your code however you wish. DO NOT include a copy of the testing script.

The graders will compile your code according to your exact specifications. They will make a reasonable effort to make it work, but if it doesn't compile, you'll receive a zero on this assignment.

Hints

I HIGHLY recommend that you develop this program directly on the server. Doing so will prevent you from having problems transferring the program back and forth and having compatibility problems.

If you do see ^M characters all over your files, try this command:

```
%dos2unix bustedFile
```

Grading

Once the program is compiled, according to your specifications, your shell will be executed to run a few sample commands against (ls, status, exit, in that order). If the program does not successfully work on those commands, it will receive a zero. If it works, it will have the testscript program ran against it (as detailed above) for final grading. Points will be assigned according to the test script.

150 points are available in the test script, while the final 10 points will be based on your style, readability, and commenting. Comment well, often, and verbosely: we want to see that you are telling us WHY you are doing things, in addition to telling us WHAT you are doing.