Program 3 – CS 344

Overview

In this assignment you will write your own shell in C, similar to bash. No other languages, including C++, are allowed, though you may use any version of C you like, such as C99. 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 smallsh. 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 (controllable by the command line and by receiving signals).

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

During the development of this program, take extra care to only do your work on our class server, as your software will likely negatively impact whatever machine it runs on, especially before it's finished. If you cause trouble on one of the non-class, public servers, it could hurt your grade! If you are having trouble logging in to any of our EECS servers because of runaway processes, please use this page to kill off any programs running on your account that might be blocking your access:

https://teach.engr.oregonstate.edu/teach.php?type=kill_runaway_processes (Links to an external site.)Links to an external site.

Specifications

The Prompt

Use the colon **:** symbol as a prompt for each command line. Be sure you flush out the output buffers each time you print, as the text that you're outputting may not reach the screen until you do in this kind of interactive program. To do this, call fflush() immediately after each and every time you output text.

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 the & character appears anywhere else, just treat it as normal text. 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 must 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.

Finally, your shell should allow blank lines and comments. Any line that begins with the # character is a comment line and should be ignored (mid-line comments, such as the C-style //, will not be supported). A blank line (one without any commands) should also do nothing. Your shell should just re-prompt for another command when it receives either a blank line or a comment line.

Command Execution

You will use fork(), exec(), and waitpid() to execute commands. From a conceptual perspective, consider setting up your shell to run in this manner: let the parent process (your shell) continue running. Whenever a non-built in command is received, have the parent fork() off a child. This child then does any needed input/output redirection before running exec() on the command given. Note that when doing redirection, that after using dup2() to set up the redirection, the redirection symbol and redirection destination/source are NOT passed into the following exec command (i.e., if the command given is `ls > more`, then you do the redirection and then simply pass `ls` into exec()).

Note that exec() will fail, and return the reason why, if it is told to execute something that it cannot do, like run a program that doesn't exist. In this case, your shell should indicate to the user that a command could not be executed (which you know because exec() returned an error), and set the value retrieved by the built-in `status` command to 1. Make sure that the child process that has had an exec() call fail terminates itself, or else it often loops back up to the top and tries to become a parent shell. This is easy to spot: if the output of the grading script seems to be repeating itself, then you've likely got a child process that didn't terminate after a failed exec().

Your shell should use the PATH variable to look for non-built in commands, and it should allow shell scripts to be executed. 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.

As above, after the fork() but before the exec() you must do any input and/or output redirection with dup2(). An input file redirected via stdin 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 (but don't exit the shell). Similarly, an output file redirected via stdout should be opened for writing 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 (but don't exit the shell).

Both stdin and stdout for a command can be redirected at the same time (see example below).

Your program must expand any instance of "$$" in a command into the process ID of the shell itself. Your shell does not otherwise perform variable expansion. This feature makes it easier to create a grading script that keeps your work separate.

Background and Foreground

The shell should wait() for completion of *foreground* commands (commands without the &) before prompting for the next command. If the command given was a foreground command, then the parent shell does NOT return command line access and control to the user until the child terminates. It is recommend to have the parent simply call waitpid() on the child, while it waits.

The shell will not wait for *background* commands to complete. If the command given was a background process, then the parent returns command line access and control to the user immediately after forking off the child. In this scenario, your parent shell will need to periodically check for the background child processes to complete (with waitpid(...NOHANG...)), so that they can be cleaned up, as the shell continues to run

and process commands. Consider storing the PIDs of non-completed background processes in an array, so that they can periodically be checked for. Alternatively, you may use a signal handler to immediately wait() for child processes that terminate, as opposed to periodically checking a list of started background processes. The time to print out when these background processes have completed is just BEFORE command line access and control are returned to the user, every time that happens.

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? Background commands should also not send their standard output to the screen: redirect stdout to /dev/null if no other target is given.

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 this 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; see example below). You will probably want to use waitpid() to check for completed background processes.

Signals

A CTRL-C command from the keyboard will send a SIGINT signal to your parent process and all children at the same time (this is a built-in part of Linux). For this assignment, make sure that SIGINT does not terminate your shell, but only terminates the foreground command if one is running. To do this, you'll have to create the appropriate signal handlers with sigaction(). The parent should not attempt to terminate the foreground child process when the parent receives a SIGINT signal: instead, the foreground child (if any) must terminate itself on receipt of this signal.

If a child foreground process is killed by a signal, the parent must immediately print out the number of the signal that killed it's foreground child process (see the example) before prompting the user for the next command.

Background processes should also not be terminated by a SIGINT signal. They will terminate themselves, continue running, or be terminated when the shell exits (see below).

A CTRL-Z command from the keyboard will send a SIGTSTP signal to your shell (this is a built-in part of Linux). For this assignment, when this signal is received by your shell, your shell must display an informative message (see below) immediately after any currently running foreground process has terminated, and then enter a state where subsequent commands can no longer be run in the background. In this state, the & operator should simply be ignored - run all such commands as if they were foreground processes. If the user sends SIGTSTP again, display another informative message (see below) immediately after any currently running foreground process terminates, and then return back to the normal condition where the & operator is once again honored for subsequent commands, allowing them to be placed in the background. See the example below for usage and the exact syntax which you must use for these two informative messages.

Built-in Commands

Your shell will support three built in commands: `exit`, `cd`, and `status`. You do not have to support input/output redirection for these built in commands and they do not have to set any exit status. These three built-in commands are the only ones that your shell will handle itself - all others are simply passed on to a member of the exec() family of functions (which member is up to you) as described above.

If the user tries to run one of these built-in commands in the background with the & option, ignore that option and run it in the foreground anyway (i.e. don't display an error, just run the command in the foreground).

The `exit` command exits your shell. It takes no arguments. When this command is run, your shell *must* kill any other processes or jobs that your shell has started before it terminates itself.

The `cd` command changes the working directory of your shell. By itself - with no arguments - it changes to the directory specified in the HOME environment variable (not to the location where smallsh was executed from, unless your shell executable is located *in* the HOME directory, in which case these are the same). This command can also take one argument: the path of a directory to change to. Your `cd` command should support both absolute and relative paths. When smallsh terminates, the original shell it was launched from will still be in its original working directory, despite your use of chdir() in smallsh. Your shell's working directory begins in whatever directory your shell's executible was launched from.

The `status` command prints out either the exit status *or* the terminating signal of the last *foreground* process (not both, processes killed by signals do not have exit statuses!) ran by your shell.

Example

Here is an example run using smallsh. Note that CTRL-C has no effect towards the bottom of the example, when it's used while sitting at the command prompt:

```
$ smallsh

: ls
junk   smallsh  smallsh.c
: ls > junk

: status

exit value 0
: cat junk
junk
smallsh
smallsh.c

: wc < junk > junk2
: wc < junk
      3     3     23
: test -f badfile
: status
exit value 1
: wc < badfile
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/0    00:00:00 sleep
```

```
 4564 pts/0     00:00:03 bash
 4867 pts/0     00:01:32 smallsh

4927 pts/0    00:00:00 ps
:
: # 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
: echo 4867

4867

: echo $$

4867

: ^C^Z

Entering foreground-only mode (& is now ignored)

: date

Mon Jan  2 11:24:33 PST 2017

: sleep 5 &

: date

Mon Jan  2 11:24:38 PST 2017
```

```
: ^Z
Exiting foreground-only mode
: date
Mon Jan  2 11:24:39 PST 2017
: sleep 5 &
background pid is 4963
: date
Mon Jan 2 11:24:39 PST 2017
: exit $
```

Grading Method

In addition to your shell needing to replicate the above example in functionality, this assignment is provided with the actual grading test script that will be used to assign your program a grade. Your program must function with this grading script, as follows. To run it, place it in the same directory as your compiled shell, chmod it (`chmod +x ./p3testscript`) and run this command from a bash prompt:

```
$ p3testscript 2>&1
```

or

```
$ p3testscript 2>&1 | more
```

or

```
$ p3testscript > 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, like put output about terminating processes further down the script than you intended. Use the script to prepare for your grade, as this is how it's being earned.

Note that as an extra challenge, no "clean run" script is provided for Program 3: you'll need to interpret the results of your program yourself.

If your program does not work with the grading script, and you instead request that we grade your script by hand, we will have to apply a 50% reduction to your final score. Make sure you work with the grading script on our class server from the very beginning!

Hints

It is recommended that you program the built-in commands first, before tackling the fork(), exec(), waitpid() specifications.

Don't forget to use fflush(stdout), as described above!

As stated above, make sure you work with the grading script on our class server from the very beginning - don't leave this to the end!

Re-Entrancy

A topic we haven't covered much is the concept of re-entrancy (Links to an external site.)Links to an external site.. This is important when we consider that signal handlers cause jumps in execution that cause problems with certain functions.

For our purposes, note that the printf() family of functions is NOT re-entrant. In your signal handlers, when outputting text, you must use other output functions!

Where to Program

I ***HIGHLY*** recommend that you develop this program directly on our course server. Doing so will prevent you from having problems transferring the program back and forth, and having compatibility problems. You have been warned: it will not behave the same on your own computer!

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

```
$ dos2unix bustedFile
```