CS 481/ECE 437 Lab 2: Simple Shell

Due Tuesday, March 1 at 08:00

This assignment is to be done individually. You may discuss this lab with others but you may not share code: each individual must turn in their own original work. In general, referencing code from the text or web that shows how to use a function is okay. However, "borrowing" code that directly implements algorithms necessary to complete portions of this lab are not. When in doubt, ask!

Concepts & Tools

- process creation (`fork()`)
- process termination (`wait()`)
- command execution (`exec()`)
- basic file I/O (`open()/close()`)
- input/output redirection (`dup2()`)
- inter-process communication (`pipe()`)

Summary and Background

A shell is a command line interpreter that allows you to execute commands and direct their input and output. A typical shell supports the basic operators, &, <, >, >>, and |, described below (as well as many more complex features that you won't have to implement like job control, command histories, regular expressions, word auto-completion.) The basic operations of a shell are:

1. print a prompt
2. read a line of input
3. parse the input command line
4. create child process(es) to execute the specified commands
5. wait for the executed commands to complete (or not)
6. repeat

In this lab you will implement a simple shell that supports the five basic operators. The assignment is divided into three parts:

1. foreground and background command execution
2. file I/O redirection
3. process I/O redirection

Each part of this lab builds on the previous: the features implemented in each part of this lab are a proper superset of the features implemented in previous sections.

General Tips:

- While the individual concepts and functions used in this lab are straightforward, start early to allow enough time to fix and debug the issues you will run into.
- As with any coding project, before writing any code, you should have a sufficient understanding of the concepts and mechanisms you will be using. You will find it beneficial to design first, then implement (followed by re-design and re-implementation as necessary). Mixing these phases will lead to wasted time, wasted effort and frustration.
- Become familiar with new library and system calls by implementing simple programs – often relevant code snippets are given in the text or class. It is easier to do this in small, simple programs than in large, complex ones.
- Most program code is written once and read many times. While you will not necessarily be graded on code style, you should always practice good, uniform styling and code commenting. This will benefit you as much as it will benefit others trying to read your code. Also, you may earn partial credit for code that does not work correctly, if we easily can understand what you were trying to do. Conversely, you may lose partial credit if your code is incredibly hideous.
- `aux_files.tgz` contains all auxiliary files referenced in this handout including my solution executable. When in doubt, your program should do what this does. Of course, if you find any bugs in this code, let me know. **Like my final program, each subsequent part should maintain all the features of the previous parts.**

Part 1: A basic shell

In Part 1, you will implement a `simsh1`:

- Your shell's prompt should be "simsh: ".
- Your shell should use execvp() to search the environment's PATH variable to find the executables for the specified commands.
- `simsh1` will support the & operator, which places the executed command line into the shell's background.

  If a command line ends with an ampersand (&), then simsh1 should not wait for the command to finish before returning the prompt. Otherwise, it should wait for the command to finish.
- You may assume that an & that is not at the end of a line is an error and print: "operator & must appear at end of command line".
- Your shell should exit when the user types CNTL-D (which generates an end-of-file or EOF) or exit.
- When using wait() wait for a command executing in the foreground, that function may return prematurely due to the termination of another child previously executing in the background. You must handle this case properly; i.e. your shell must wait for the command executing in the foreground to exit before it prints the prompt and fetches another command line.
- Your shell should use wait(), waitpid() or even wait3() to clean up its zombie processes, children processes that have exited but have not been waited for.
- For each part of this lab, make sure you handle all error conditions you can think of, including operator misuse (like trying to redirect output to a pipe and a file at the same time) or bad commands.

Tips

- You may find the code in chop_line.h and chop_line.c to chop your command line into individual tokens. Usage information is in the header file.
- You may also find the simple linked list implementation in list.h and list.c useful.
- Assume that a whitespace separates commands, arguments and shell operators.
- I found it useful to validate the command line before trying to parse and execute the specified commands.
- You may limit the length of a command line to 4,096 characters.
- You may limit the number of arguments to a single command to 32. (Note this is not the total number of arguments on the entire line: a command line may be comprised of multiple commands.
- During the debugging phase, look up and use the `ps` and `kill` commands to find and kill stray processes left around by your shell.

Write-up Questions

1. What is a shell built-in command?
2. What are environment variables? Give five examples along with their uses.

Part 2: I/O File Redirection

In Part 2, you will implement `simsh2.`, which supports the redirection of command input and output from/to files – i.e. <, >, and >> operators.

- `simsh: cmd < infile`
  - run cmd redirecting standard input from infile
  - an error occurs if infile does not exist
- `simsh: cmd > outfile`
  - run cmd redirecting standard output to outfile
  - outfile should not exist; it must be created
  - an error occurs if outfile exists
- `simsh: cmd >> outfile`
  - run cmd redirecting standard output to outfile
  - outfile may or may not exist
  - if outfile exists, output should be appended to the end of this file
  - if outfile does not exist, it should be created

You will need to use dup2() to redirect the STDIN_FILENO and STDOUT_FILENO file descriptors. As appropriate, simsh2 should print the following error messages:

- Ambiguous input redirect.
- Ambiguous output redirect.
- Missing name for redirect.
- Invalid null command.
- <filename>: No such file or directory.
- <filename>: File exists.

For the last two error messages, <filename> is the name of the erroneous file.

Tips

- Whenever a forked child is about to call execvp(), it should only have three file descriptors opened: 0, 1 and 2.

Write-up Questions

3. What difference (if any) does it make if open() and dup2() are called before versus after the child process is forked()?

Part 3: Pipes

Finally, in Part 3 you will implement `simsh3`, which adds support for the pipe operator, |. You will be using the `pipe()` system call to add this feature.

Tips

- Any number of processes can be pipelined together, but as with the rest of this assignment, develop incrementally.
- When multiple commands are pipelined on a single command line, unless the & operator is used, your shell must wait for each command in the pipeline to complete before returning to the command prompt.

What to turn in

YOU MUST FOLLOW THESE INSTRUCTIONS PRECISELY.

You will turn in (at least) 5 files:

- `simsh1.c` from Part 1
- `simsh2.c` from Part 2
- `simsh3.c` from Part 3
- all auxiliary source or header files necessary to build your programs, including any you've borrowed from the lab's write-up.
- a make file named, `Makefile`, which builds the `simsh1`, `simsh2` and `simsh3` programs. `write-up.txt` containing your write-up answers.

Do not turn in any any object, binary or core files. When you are ready to turn in your assignment:

- Place the requisite files in a directory named lastname_lab2 where lastname is your last name. move to the parent directory that contains this lab2 directory; execute the command:
  - `tar -czf <lab2_dir>.tgz <lab2_dir>`
  - where <lab2_dir> is the name of the directory containing your lab 2 files. This will create a new file <lab2_dir>.tgz containing the contents of your <lab2_dir>. You can verify the contents of this "compressed tar file" with the following command:
  - `tar -tzf <lab2_dir>.tgz`
- Submit the file `<lab2_dir>.tgz` via UNM Blackboard.