CIS 3110: Operating Systems

Assignment 1: Making Your Own Shell

Due Wednesday, Feb. 6, 2019 @ 11:59pm

In this assignment you are required to implement a simple Linux shell program in C. This assignment focuses on processes, a central building block of any modern-day operating system (OS). A process is an instance of a running computer program. Any modern OS provides an interface for computer users to run programs. The most popular yet simple way is using a shell, which is simply a program that conveniently allows you to run other programs. On “Unix-like” operating systems such as Linux, a shell can be invoked through the shell command in the command line interface (CLI). There are many different shells in “Unix-like” OS. Among them, the top most used open source shells are sh (the Bourne shell), csh (the C shell), and bash (the Bourne Again shell). Read up on your favorite shell to see what it does. Note that a modern OS provides graphical user interfaces (GUIs) in addition to command line interfaces (CLIs) such as the shell.

The facilities of the operating system available to interactive users can be made by a shell through a list of commands, aka shell commands. These shell commands can be divided into two categories, built-in commands and external programs. Built-in commands refer to the commands whose implementations are built directly into the shell program, whereas external programs as its name indicates, refer to these programs which exist in separate executable files, like *ls*, which is used to list directory contents of files and directories.

Simply put, a shell is a command language interpreter. Specifically, when started, it prompts for user input. It takes commands entered by a user from the keyboard. The shell first checks if the command entered by the user is a built-in command or not. If it is, the shell simply invokes the built-in command implemented in the shell. Otherwise, the shell searches for the executable file in a set of directories, which is usually defined by the PATH environment variable. If the executable program is found in a directory, the shell will execute the program. Otherwise, an error message, for example, “No command 'lsd' found”, where *lsd* is the wrongly typed command, will display. When the shell executes the command, it first forks a process, which is called child process. The child process will load and run the program, for example, by using exec call.The shell waits for the child process to complete before displaying the prompt again, waiting for the next command. It will continue doing so until the user types exit to close the shell. The following is an example of using bash to run the *ls* command

```
# ls
Desktop
#
```

Where the pound sign # is the shell prompt.

The requirements for completing this assignment successfully are described under **specification**.

Specification

(1) Shell prompt

When your shell starts, it should display a shell prompt that contains your user name and the name of the machine followed by a dollar sign. Something like this:

```
[student@ubuntu]$
```

where student is the user name and Ubuntu is the name of the machine. Note that if you login as the superuser or root, the last character of your shell prompt is # rather than $.
System calls: getuid(), gethostname()

Note: The system call getuid() only deals with user IDs. You need to find a way to convert the user ID to the user name.

(2) Your shell must support the following:

1. **The internal shell command "exit" which terminates the shell**.
   Concepts: shell commands, exiting the shell
   System calls: exit()
2. **A command with no arguments**
   Example: ls
   Details: Your shell must block until the command completes and, if the return code is abnormal, print out a message to that effect.
   Concepts: Forking a child process, waiting for it to complete, synchronous execution
   System calls: fork(), execvp(), exit(), wait()
3. **A command with arguments**
   Example: ls -l
   Details: Argument 0 is the name of the command
   Concepts: Command-line parameters
4. **A command, with or without arguments, executed in the background using &.**
   For simplicity, assume that if present the & is always the last thing on the line.
   Example: xemacs &
   Details: In this case, your shell must execute the command and return immediately, not blocking until the command finishes.
   Concepts: Background execution, signals, signal handlers, processes, asynchronous execution
   System calls: sigset()
5. **A command, with or without arguments, whose output is redirected to a file**
   Example: ls -l > foo
   Details: This takes the output of the command and put it in the named file

Concepts: File operations, output redirection
<u>System calls</u>: freopen()

6. **A command, with or without arguments, whose input is redirected from a file**
   Example: sort < testfile
   Details: This takes the named file as input to the command
   Concepts: Input redirection, more file operations
   <u>System calls:</u> freopen()

(3) Your shell should implement the following new commands:

1. **A command "gcd" that finds the greatest common divisor (gcd) of two given numbers in command line (decimal or hexadecimal).**
   Example: gcd 9 0xc
   Details: Find the gcd of two given numbers and output and output the gcd
   Output: GCD(9, 0xc) = 3
2. **A command "args" that counts and lists command line arguments.**
   Example: args 1 2 3 4 5 "six, seven"
   Details: count and list arguments
   Output: argc = 6, args = 1, 2, 3, 4, 5, "six, seven"
3. **A command "???" that is named and designed by yourself.**
   Example: you make an example
   Details: you specify the function
   Output: you give a sample output

Note: You must check and correctly handle the return values of Every system call. This means that you need to read the man pages for each function to figure out what the possible return values are, what errors they indicate, and what you must do when you get that error.

Also, as for the above new commands, **your implementation should be robust for bad arguments**! If the argument is invalid or bad, your program should print a usage statement, and then exit graceful indicating an unsuccessful termination[1]. Further, your program should be robust to handle abnormal situations, for example, reading non-existing files if your command needs to read a file.

There are several parts you must pay attention to about usage statements[2]

- **The usage message**: it always starts with the word "usage", followed by the program name and the names of the arguments. Argument names should be descriptive if possible, telling what the arguments refer to, like "filename" in the example below. Argument names should not contain spaces! Optional arguments are put between square brackets, like "-l" for the Linux command *ls*. Do not use square brackets for non-optional arguments! Always print to stderr, not to stdout, to indicate that the program has been invoked incorrectly.
- **The program name**: always use argv[0] to refer to the program name rather than writing it out explicitly. This means that if you rename the program (which is common) you won't have to re-write the code.

- **Exiting the program**: use the exit function, which is defined in the header file <stdlib.h>. Any non-zero argument to exit (e.g. exit(1)) signals an unsuccessful completion of the program (a zero argument to exit (exit(0)) indicates successful completion of the program, but you rarely need to use exit for this). Or, you can simply use EXIT_FAILURE and EXIT_SUCCESS (which are defined in <stdlib.h>) instead of 1 and 0 as arguments to exit.

For example, the following is a code snippet for the gcd command, which prints a usage statement, and then exits the program by indicating an unsuccessful termination

```
fprintf(stderr, "usage: %s number1 number2\n", argv[0]);
exit(EXIT_FAILURE);
```

Skeleton

For your reference, the code skeleton is provided as a starting point for your shell program.

- The file ish.c contains the skeleton for your shell program
- The file lex.c contains the skeleton for parsing the input line using lex. Lex is used for compiler construction. For more information:
  http://epaperpress.com/lexandyacc/thl.html

Your task is to complete the above skeleton code with the appropriate c code to write your own simple shell. Besides the functionalities required in this assignment, an emphasis should also be placed on writing concise easy to read code. Please refer to the coding style guidelines (e.g., C Programming Style Guide[3]) to provide direction regarding the format of the code you write. As the program grows, you may want to improve the structure of the code by moving certain parts into separate functions.

Note: In order to compile your source code in simple and fast way, you are required to create a Makefile for building the executable file. A Makefile is a recipe for the make utility[4] how to create some file (called a target) from some other files (called dependencies) using a set of commands run by the shell. Please refer to the following tutorials on how to use make and write Makefiles

- Using make and writing Makefiles
  https://www.cs.swarthmore.edu/~newhall/unixhelp/howto_makefiles.html
- Makefile Tutorial
  https://www.wooster.edu/_media/files/academics/areas/computer-science/resources/makefile-tut.pdf
- Writing Make Files
  https://www.cs.bu.edu/teaching/cpp/writing-makefiles/

Submission

- If you have any problems in the development of your programs, contact the teaching assistant (TA) assigned to this course.
- You are encouraged to discuss this project with fellow students. However, you are **not** allowed to share code with any student.
- Each student should submit a brief report (2-3 paragraphs) that explains your algorithm or solution strategy, any assumptions you made, and the way to run your program.
- If your TA is unable to run/test your program, you should present a demo arranged by your TA's request.
- Please only submit the source code files plus any files required to build your shell program as well as any files requested in the assignment, including
  - the complete source code;
  - the Makefile; and
  - a writeup that explains your algorithm or solution strategy, any assumptions you made, and the way to run your program.
- How to name your programming assignment projects: For any assignment, zip all the files required to a zip file with name: CIS3110_<assignment_number>_XXX.zip, where <assignment_number> is the assisgnement number you are solving (e.g., a1 for Programming Assignment 1) and XXX is your University of Guelph's email ID (Central Login ID). This naming convention facilitates the tasks of marking for the instructor and course TAs. It also helps you in organizing your course work. Failure to follow the requirements will result in mark reduction.

  Note: to zip and unzip files in Unix:
  ```
  $zip -r filename.zip files
  $unzip filename.zip
  ```

References:
[1] Exit Status.
https://www.gnu.org/software/libc/manual/html_node/Exit-Status.html
[2] Processing command-line arguments.
http://courses.cms.caltech.edu/cs11/material/c/mike/misc/cmdline_args.html
[3] C Programming Style Guide
http://faculty.cs.tamu.edu/welch/teaching/cstyle.html
[4] http://man7.org/linux/man-pages/man1/make.1.html