COSC 40203 - Operating Systems

Project 1: Create Your Own Shell

Spring 2017
Due: 23:59:59, February 13th, 2017

Objective

The purpose of this assignment is to get everyone up to speed on system programming and to gain some familiarity with system calls and process creation. A secondary goal is to use some additional software development tools provided in the UNIX environment. In this assignment you are to implement a UNIX shell program. A shell is simply a CLI program that conveniently allows you to run other programs. Read up on your favorite shell (such as bash, csh, tcsh, sh, ...) to see what it does. This project is based on the shell project in your textbook in Chapter 3, and a custom project of my own design.

Getting Started

Download the `ShellLab.zip` file from TCU Online to your local machine. Upload the file to `thompson.cs.tcu.edu`. Extract the ShellLab subdirectory by typing `unzip ShellLab.zip` at the command line. The subdirectory created is where you will do all of your work. A basic REPL algorithm has been provided to you. Your program extend this baseline sourcecode with internal and external shell commands and process control.

You need to create a `Makefile` symbolic link in your ShellLab subdirectory. This link will point to the language specific makefile you wish use for this project. If you choose"C" then type `ln -s Makefile_c Makefile`. If you choose "C++" then type `ln -s Makefile_cpp Makefile`. From there on you will edit either the tcush.c or tcush.cpp file respectively. To compile your project, simply type `make`.

The extra files that are supplied or get created perform the scanning functionality for a input line into your shell. This project uses some compiler tools called "flex" that will scan the input line and separate it into tokens - character strings. Specifically the file `scan.l` describes how to split the line. It may look cryptic, but you do not have to make any changes to this file at all.

Whether you choose "C" or "C++" a separate program template have been provided for you (you pick). Inside each of these programs is a simple main function that calls the `gettoks()` function and returns an array of c-strings (null terminated) where each array element is one token (character string). The array of strings is then printed out in a simple for-loop. You can use the array of strings (tokens) to determine the name of the program to run, command line arguments or options, and special characters such as pipes and I/O redirection symbols.

Initial Project Setup

1. Download the program template (`ShellLab.zip`) from Pearson Learning Studio. Upload or copy this zip file to your account on `thompson.cs.tcu.edu`.
2. Unzip the project: `thompson> unzip ShellLab.zip`. This will create a subdirectory named `ShellLab` with several files inside. Go into that subdirectory.
3. Get familiar with the files that were uncompressed. Select a programming language for your project; either C or C++. There is not much difference between the two. C++ does have several nice predefined C++ standard containers such as list, stack and queue that might be useful. These are defined in the C++ standard library.
4. Create a symbolic link to your makefile: `thompson> ln -s Makefile_c Makefile`. Or change it to `Makefile_cpp` for the C++ version.
5. Build your project: `thompson> make`
6. Run the shell program. `thompson> ./tcush`
7. To exit the shell, enter `myexit`

Requirements

1. (2 points) Ignore signals SIGINT, SIGTSTP, and SIGQUIT. This some of the first executable statements in the `main()` function.
2. (3 points) Create a dynamic shell prompt that changes either every time the user presses [ENTER] to execute a command, or changes as a result of the last command. Examples include showing the current date and time, showing the current working subdirectory, command number, etc. Static prompts will result in loss of points. Creating a new shell prompt should be a separate function.
3. (25 points) If the command entered is an external command, then your shell will create a child process with or without arguments. The child process will run in the foreground by default. If the last token on the command line is the ampersand character (&), then your program must fork a child process and execute in the background. If it is a background process, your shell will print the next prompt and wait for the user to enter the next command. You should have separate parent and child functions in your program.
   - Example foreground: `thompson> cal 2016`
   - Example background: `thompson> emacs tcush.cpp &`
4. (5 points) A command, with or without arguments that uses file redirection.
   - Example: `thompson> ls -l > out.txt`
   - Example: `thompson> sort < in.txt`

- Example: `thompson> sort -r < in.txt > out.txt`

5. (5 points) A command with or without arguments that pipes the output of the first process into the input of a second process.
   - Example: `thompson> ls | grep README`
   - Example: `thompson> sort -r < tcush.c > out.txt`
6. (20 points) Implement a command history feature. Follow the specification on page 159 of our textbook for a description of the history feature. This should be written as separate functions.
7. (10 points) Implement a new internal command called `forweb` which takes the name of a subdirectory *d*irname and makes all files `o+r` and all folders `o+rx` in the file hierarchy rooted at *d*irname. This should be written as separate functions.
8. (10 points) Implement a new internal command called `nls` that is similar to the `ls` command but that, by default, displays regular files and directories separately. This should be written as separate functions.
9. (10 points) Implement a new internal command called `fil`. The usage is as follows:
   - Example: `thompson> fil [from] [to]`

   to transform text from the named file *from* to the named file *to*. If only one file argument is supplied, it is assumed to be for the *from* file. A dash means standard input; a missing *to* means standard output. The `fil` command works as follows:

   a. All tabs are replaced by an equivalent number of spaces.
   b. All trailing blanks at the end of each line are removed.
   c. All lines longer than 132 characters are folded.
   d. A form fed is added for every 66 lines from the previous form feed.
   e. All BACKSPACE and nonprinting characters are removed.

   This should be written as separate functions.

Note: there are a total of 5 internal commands: `myquit`, `history`, `forweb`, `nls`, and `fil`.

Submitting Your Project

Create a zip file of subdirectory that contains your source code and a README file. In the README file please be sure to indicate what language you used, how many hours you did for design, development, and testing, and if there are any unimplemented or non-working features. Please be sure to `make clean` prior to creating the zip file (only include source code; no object files or executables). Do not modify `scan.l`.

Assessment and Grading

This is an individual assignment. Write a well-documented, well-structured C/C++ program using procedures and functions. Programs that are all in the "main function" or have excessively long functions will result in loss of points. Your project must following the documentation standards defined as described on the course web page. You may use the STL in your solution. You may use any computing system to develop your shell program however, I will test your program on `thompson.cs.tcu.edu`. This project is worth 100 points.