CS 240 Summer 2017

Lab 5: Run-time Environment and Client/Server Programming in C (270 pts)

Due: 07/19/2017 (Wed), 11:59 PM

Objective

The objective of this lab is to make further use of system utilities when developing C programs, get familiar with the run-time environment of C programs, and start writing C client/server apps which comprise the bulk of apps in use today.

Reading

Read chapters 7 and 8 from K&R (textbook).

Lab 5 Code Base

The C code base for lab5 is available as a tarball

/u/data/u3/park/pub/cs240/lab5.tar

on the lab machines.

Problems [270 pts]

Problem 1 (70 pts)

This is an extension of Problem 1, Lab 4. Code an additional function calc_power() with function prototype the same as calc_diff(). calc_power() computes the square of the difference of two vectors: component-wise, one is subtracted from the other and then the difference is squared. For example, for two vectors (4, 5, 1) and (2, 7, 4) we have

((4-2)^2, (5-7)^2, (1-4)^2) = (4, 4, 9)

Use a math library function in <math.h> that computes powers (do not multiply a number by itself) to calculate its square.

A difference in the user interface (UI) aspect of the coding is that the input is provided not via stdin but through a file, and the app's output is not to stdout but to a file. Use the fopen(), fscanf(), fprintf(), and fclose() discussed in class to read the input from a file "vecin.dat" and write the result to a file named "vecout.dat" in the current directory.

A difference in the system aspect of the implementation is that you are asked to create a dynamic (i.e., shared) library instead of a static library used in lab4. Creation of shared libraries is more system dependent but the following is a straightforward way to do it as a user (as opposed to as root for system wide use). First, compile each source file separately with the -c option and the additional -fPIC option (PIC stands for "position independent code"). Second, to generate a shared library named libmymathlib.so, run

```
gcc -shared -o libmymathlib.so *.o
```

where for brevity the wildcard * is used for all object files. Please specify the object files separately. Fourth, linking main.o with the shared library follows the same convention as with the static library in lab4. However, don't forget that your code also uses functions from the math library. Fifth, when you try to run mymain you are likely to get an error from the loader that indicates that it cannot find the shared library functions. One way to specify to the loader where to find libmymathlib.so is via the environment variable LD_LIBRARY_PATH. Check using echo $LD_LIBRARY_PATH its value (i.e., existing paths). Unless you have already customized it, it is likely to be undefined or empty. How you define LD_LIBRARY_PATH is dependent on the shell you are running. In the case of tcsh, the following

```
setenv LD_LIBRARY_PATH <pathname-of-lib-directory>
```

does the trick. For bash or Bourne shell (sh), the following will do

```
export LD_LIBRARY_PATH=<pathname-of-lib-directory>
```

Depending on what shell you are running, configure LD_LIBRARY_PATH accordingly so that the loader can find it. If all goes well, your mymain executable should be able to access your shared library functions in libmymathlib.so dynamically when it is running (as well as those in the math library). Do the above manually once so that you know how to do it step-by-step.

As in lab4, automate the compilation and linking process with a dynamic libary process by using Makefile. The format for specifying dependencies and actions in Makefile apply as before. However, insert the statement

```
all: libmymathlib.so mymain
```

at the top of Makefile which indicates that you are generating two targets (with dependencies): libmymathlib.so and mymain. You can define LD_LIBRARY_PATH from the shell or inside Makefile; it is up to you. Submit your work in v10/.

Problem 2 (100 pts)

Print to stdout a generic shell prompt "$ " and wait for user input (a string) by calling getchar(). The input entered by the user is treated as a command to a generic Linux shell (e.g., tcsh, bash) and executed using the function system(). The app you are writing can be viewed as your customizable shell that takes commands (binary executables or interpreted shell commands) as input and executes them. As such, your code must follow the structure of an infinite loop within which user input and execution of user requested command is performed using system(). This is the design of a canonical client/server app. Code, compile, and test the shell, call it esh, and verify that it works correctly.

To make it a tad more useful, add the following custom features to esh:

First, when the user enters "cprt <string>" then make esh change its prompt from its default "$ " to whatever <string> has been entered followed by colon and space.

Second, if the user hits the return key without providing any input, go to a new line, print the prompt, and wait for user input.

third, if the user enters "E", "e", "X", or "x", then esh should exit, that is, terminate. Note that when esh terminates, your terminal returns to the control of the shell program under which you ran esh.

Fourth, when the user enters "lck" then lock the terminal by printing the string "key to unlock: " on stdout and waiting on a secret string to be entered to unlock. Set the secret string by reading it from a file secretstring.txt when esh starts up. If the secret string is correctly entered, unlock by printing the prompt and resuming normal operation. On any other input, print "key to unlock: " on a new line and keep waiting for the correct secret string to be entered.

Fifth, come up with two of your own customization and describe what they do in lab5.pdf. Try to devise custom shell commands that you would actually like to use on a regular basis. Submit your code in v11/.

Problem 3 (100 pts)

Re-architect the esh shell of Problem 2 to replace system() with system calls to Linux to spawn a child using fork() and execute a command using execlp() which invokes the loader. After reading user input from esh's prompt, if the input is determined not to be one of the shell commands, then a call to fork() is made to create a child process that calls execlp() to run the requested command. To use execlp() include the header file <unistd.h>. As an example, to execute the binary ls from within the binary esh, the child process calls

```
execlp("ls","ls",NULL);
```

which invokes the Linux loader to load the "ls" binary into main memory and creates a new process that executes ls. Before calling execlp() with the user provided binary as the first and second arguments to execlp(), print to standard output using fprintf() a message

"new process will run: <name-of-binary>"

where <name-of-binary> is the name of the binary entered by the user. For simplicity, assume arguments to binaries are not allowed (i.e., "ls" is fine but "ls -l" is not). Shell commands (e.g., cprt, lck) should be carried out by the parent as in Problem 3 without forking a child process. Code, compile, and test the modified shell, call it eesh. Submit your code in v12/.

Bonus Problem (15 pts)

Consulting the man page of bash, create a man page for eesh. It does not need to be as long and comprehensive. Focus on the features and limitations of eesh so that a user who reads your man page will know how to use it. Include the man pages in lab5.pdf.

Turn-in Instructions

Electronic turn-in instructions:

i) For code submissions, follow the directions specified in the problems.

ii) Use turnin to manage lab assignment submissions. Rename lab5 to mylab5. Go to the parent directory of your mylab5 directory and type the command

```
turnin -v -c cs240 -p lab5 mylab5
```

Please note the assignment submission policy specified in the course home page.

Back to the CS 240 web page