Advanced Exercises Set 10
Libraries

EECS 201 Winter 2020 COVID-19 Edition

Submission Instructions

To receive credit for this assignment you will need to email the staff at eecs201-staff@eecs.umich.edu with `tar` archive(s) of your submissions attached as well as any additional information requested in the body of your email. You may choose the exercises that you wish to do from this set. Each exercise is denoted by its point count. **Extra credit is given for early turn-ins of advanced exercises. These details can be found on the website under the advanced homework grading policy. For these email-based submissions, your latest submission of the `tar` archive/questions will determine your extra credit.**

Preface

For these exercises you're going to need to be in a Linux environment, be it WSL or your Ubuntu VM or whatever else you have.

There are also some provided example files.

```
curl -O https://www.eecs.umich.edu/courses/eecs201/files/assignments/adv10-files.tar.gz
tar xzf adv10-files.tar.gz
```

1 Let's have some fun: ncurses (5)

ncurses is a library that provides an abstraction layer to make it easier to build user interfaces in a terminal. It's used in many different applications that do more with the terminal than print lines, such as text editors (e.g. Vim, Emacs) or games (NetHack).

For this exercise you'll be playing around with linking against a library and learning how to use it by making a C/C++ application that makes use of ncurses.

What I want you to do is write a C/C++ application that draws a character (your choice) on the terminal, which can be moved with vi/Vim style navigation (h/j/k/l for left/down/up/right) and quits when 'q' is pressed. In addition, create a Makefile that builds the application. Feel free to add onto it, or, if you want to do something else, ask the staff for permission.

The package for ncurses on Ubuntu is `libncurses5-dev`.

A nice resource is the HOWTO on it.

- ☐ In your tarball include the source code for your application and a Makefile to build it.
- ☐ In the body of your email let me know if you've used ncurses before and your general thoughts about this exercise.

2 Let's have some fun: SDL2 (10)

SDL2 (Simple DirectMedia Layer) is a library that offers cross-platform abstractions for video-game and other media related hardware such as keyboard, audio, and graphics. The package for SDL2 on Ubuntu is `libsdl2-dev`.

In a similar manner to #1, you'll be playing around with SDL2 to make a C/C++ application.

Following the theme for #1, I want you to write a C/C++ application that uses SDL2 that draws a rectangle (your choice of color) that moves around using the arrow keys.

Here is the official reference for SDL2. Here is a tutorial for SDL2. Particularly relevant for this exercises are the Event Driven Programming, Key Presses, and the Geometry Rendering lessons.

- ☐ Include this report in the body of your email.
- ☐ In the body of your email let me know if you've used SDL2 before and your general thoughts about this exercise.

3 Runtime library linking (10)

A closely related topic to the idea of dynamic/shared libraries is the idea of dynamic linking at *runtime*.
Normally when you run a program, the dynamic libraries that its linked against would be loaded at program *load time*. While doing it at load time more dynamic than say, static linking, you can be even more dynamic by loading libraries at runtime!
On POSIX systems we have the `dl` library that provides the ability to run the dynamic linker to load libraries at runtime (on Windows there's a similar function called `LoadLibrary`).
With the ability to load code at runtime, we can implement the idea of plugins! In this exercise, we'll be filling out the framework for a plugin system.
Files are provided in the archive under `plugin`. How this is structured is that `runner.c` is compiled into `runner` and serves as the "driver" program that loads plugins. Plugins are merely dynamic shared objects: `*.so` files. There's a Makefile target that turns `plugin*.c` files into `plugin*.so` files. The plugins for this application follow a standard where they must implement three functions: `init()`, `run()`, and `cleanup()`.

Your job is to utilize `dlopen`, `dlsym`, and `dlclose` (they have manpages!), as well as a bit of function pointers, to implement the TODOs in the `runner.c` code.

If you have not taken EECS 370, you may be unaware of what a "symbol" is. A "symbol" is an identifier for some resource, such as a global variable or a function. In the context of binary executables and libraries, these symbols are associated with an address in memory where the resource is. For instance, the symbol "init" for one of the plugins may resolve to address `0x4100` and the symbol "cleanup" may resolve to `0x4200`. By getting symbols from a library you're effectively locating where the resource is so you can use it (via the magic of pointers!).

- ☐ Include your modified `runner.c` in your tarball.
- ☐ In your email, mention if you have taken EECS 370 before and if you knew what symbols were before doing this exercise.