Advanced - Libraries

EECS 201 Fall 2025

Submission Instructions

This assignment will be submitted as a repository on the EECS GitLab server (see Basic - Git 1). Create a private, blank, **README-less (uncheck that box!)** Project on it with the name/path/URL `eecs201-adv-lib` and add `mmdarden` as a Reporter. The submission branch will be `plugin`. If this branch is not already the default initial branch, you initialize the local repo with an additional argument: `git init --initial-branch=plugin` or `git init -b plugin` if your version of Git is recent enough. Otherwise you can create a branch with this name after your first commit.

To finish the submission process and get your grade, you will need to run the autograder. You will need to SSH into the course server (see Basic - Intro):
`local$ ssh uniqname@peritia.eecs.umich.edu`
(the `local$` referring to a shell prompt on your system)

and then run `eecs201-test`:
`peritia$ eecs201-test adv-lib`
(the `peritia$` referring to a shell prompt on the server)

Preface

In this assignment you'll be provided yet another zipped archive containing some starter files.

`https://www.eecs.umich.edu/courses/eecs201/fa2025/files/assignments/adv-lib.tar.gz`

1 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 assignment, we'll be filling out the framework for a plugin system.
Files are provided in the extracted archive. 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: `plugin_init()`, `plugin_run()`, and `plugin_cleanup()`. They also must have two strings for identification: `plugin_name` and `plugin_version`.

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 `"plugin_init"` for one of the plugins may resolve to address `0x4100` and the symbol `"plugin_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!). Helpful hints:

- Note that `dlsym()` returns a pointer to your queried resource. Think about how that interacts with resources that are themselves pointers like `const char *`. For example, a resource that is an `int` will have a pointer to it returned (an `int *`). What happens if the resource is a `const char *`? Playing with pointers is half the fun of C and C++ ;)

1. Initialize a Git repository in the extracted `adv-lib` directory with an initial branch of `plugin`. If your Git version does not support this, create the `plugin` branch after you make your first commit.
2. Create a **private** project named `eecs201-adv-lib` on the EECS GitLab (`gitlab.eecs.umich.edu`) add the instructor `mmdarden` as a **Reporter**. Set this EECS GitLab project as your remote: you'll be pushing to it in order to submit.
3. Modify `runner.c` and implement the required behavior.
4. Fill out the `report.txt` file in the following steps:
5. On the first line provide an integer time in minutes of how long it took for you to complete this assignment. It should just be an integer: no letters or words.
6. On the second line and beyond, write down what you learned while doing this assignment. If you already knew how to do all of this, put down "N/A".
7. Commit your `report.txt` file and push your commits to your remote.
8. Run the autograder!