Note: Please ssh to Watson 132 for this activity!

1 Modules

The `module.c` file shows how to correctly implement a modular C program. Here are the compilation steps.

```
$ clang -c superutilA.c
$ clang -c superutilB.c
$ clang -c module.c
$ clang -o modularProg superutilA.o superutilB.o module.o
```

After these steps, you can run `modularProg`.

TODO: Verify the following concepts by inspecting the files above.

- We call the pair "file.c" and "file.h" a *module*.
- We call "file.c" the *implementation file* of the module, and "file.h" the *header file* of the module.
- The header file *externalizes* some (but not necessarily all) variables and functions from the implementation file.
- Variables declared `static` inside a implementation file are **private** to the module. If you have two implementation files declaring a variable with the same name, both non-static, the program cannot be put together ("linked").
- Note that non-static variables can be externalized to other modules by noting that variable in the header file with the keyword `extern`.

2 Static Linking

A static library is a collection of object files (that is, pre-compiled code) that can be *statically linked* into a program. In other words, the code that composes the library is concatenated with the generated program image at compile time.

The steps necessary to create a static library are given below. First, we create a file called `libsuperutil.a`, containing all the object files and an index. The index maps symbols – names of exported functions and variables – of all the symbols found in `superutilA.o` and `superutilB.o` into the location of their appropriate code (text) in the file `libsuperutil.a`.

The tool `ar` is used to create this structure.

```
$ ar rs libsuperutil.a superutilA.o superutilB.o
```

We move our static library into the "Libraries" directory, and the include files, that *are still required in order to compile individual files* into the "Includes" directory. Normally, these files reside in `/usr/lib` `/usr/include`, respectively.

```
$ mkdir Libraries Includes
$ mv libsuperutil.a Libraries/
$ cp superutil.h Includes/
```

Now we will try to compile our program. We pass the directory where `clang` needs to look for includes, but we still have to tell it where to locate the static library!

```
$ clang -IIncludes -c slib.c
$ clang -o slib slib.o
Undefined symbols for architecture x86_64:
  "_functionA", referenced from:
      _main in slib.o
  "_functionB", referenced from:
      _main in slib.o
  "_visible", referenced from:
      _main in slib.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

We have to give two pieces of information to `clang`:

1. The `-lsuperutil` flag, indicating that it should look for code in a library called `libsuperutil`;
2. The `-LLibraries` flag, indicating the location of the libraries directory where to search for `libsuperutil`.

```
$ clang -o slib slib.o -lsuperutil
ld: library not found for -lsuperutil
clang: error: linker command failed with exit code 1 (use -v to see invocation)
$ clang -o staticProg slib.o -LLibraries -lsuperutil
```

You can now run `staticProg`; this binary is equivalent to `modularProg`. It contains code and data from `libsuperutil`.

Static linking can be interesting because the generated program has no runtime dependencies on code from the "SuperUtil" library. *All* the functions from the static library have been incorporated into the executable – not only the *used* functions, but all of them.

Discussion Question

Give one advantage and one disadvantage of using static linked libraries.

3 Dynamic Linking

With dynamic linking, we pack object code that is *loaded* into processes when they are created. To create a shared library, we have to pass the `-shared` flag to `clang`. To make the relocation efficient, we need to generate *position independent code*. This is done using the `-fPIC` in the compile line.

```
clang -c -fPIC superutilA.c
clang -c -fPIC superutilB.c
clang -fPIC -shared -o libsuperutil.so superutilA.o superutilB.o
mv libsuperutil.so Libraries/
clang -IIncludes -c slib.c
clang -o dynamicProg slib.o -LLibraries -lsuperutil
```

If you try to run `dynamicProg`, you'll get an error. When we run a dynamically linked program, the loader will look in standard directories for shared libraries (like `/usr/lib` or `/lib`), but if you have a different directory where you stash your shared libraries, you have to tell the loader to look at that location. You do that by using `export` to set a shell's *environment variable* called `LD_LIBARY_PATH`, which has a syntax similar to the `PATH` environment variable. (`DYLD_LIBARY_PATH` on macOS.)

```
echo $LD_LIBRARY_PATH
export LD_LIBRARY_PATH=Libraries:$LD_LIBRARY_PATH
echo $LD_LIBRARY_PATH
./dynamicProg
```

Bonus Activity

Inspect `/usr/lib`. Note that the system's libraries are versioned, and there's a symbolic link to the default version to be used for each library. What libraries do you recognize?

Run the `ldd` program on your `dynamicProg` executable. This program will identify all the libraries it depends on.

1. Can you identify the loader and the SuperUtil library?
2. Can you identify a dependency for the C library? If so, then the linker must have been invoked with "-lc", but this is almost always needed, so it's done automatically unless you give clang the -nostdlib argument.

Now, here's something fun: the loader itself is a library that loads all the other libraries! The loader in Linux is called `ld-linux.so`; on Free-BSD it's called `ld-elf.so`. A quote from the manpages:

> The `ld-elf.so.1` utility itself is loaded by the kernel together with any dynamically-linked program that is to be executed. The kernel transfers control to the dynamic linker. After the dynamic linker has finished loading, relocating, and initializing the program and its required shared objects, it transfers control to the entry point of the program.

In other words, when the program is loaded, the OS gives control to some code in `ld-linux` instead of jumping straight to `main` (or `start_`). The responsibility of the code in `ld-linux` is to load all shared libraries and only then jump to the `main` (or `start_`) method.

The loader uses the system calls `dlopen` and `dlsym` to load functions. If you are planning to implement a loader, look at the documentation for those syscalls.

Here's much more information that you'll never need – unless you are writing a dynamic loader for a kernel.

`http://eli.thegreenplace.net/2011/11/03/`
`position-independent-code-pic-in-shared-libraries/`