Exercise 13
Shared, Dynamically Linked Libraries

For this exercise, you're going to try out using shared, dynamically linked libraries on Linux. You'll write your own little shared library and see how a system can be configured to use the latest compatible release of a dynamically linked library without re-compiling client code.

You'll need to do this exercise on one of the EOS Linux machines. I've already compiled some of the components you'll need on these systems. They may work on other Linux machines (they did for me), but it's possible you could run into compatibility problems on different systems. Plus, you're going to be submitting a compiled binary for this exercise. Building it on an EOS Linux machine will help make sure your solution will work for me when I try it out.

Once you're logged in on an EOS Linux machine, make a directory where you expect to work. Change to that directory and use the following curl commands to copy down the files you'll need.

```
curl -O https://people.engr.ncsu.edu/dbsturgi/246/exercise/exercise13/driver.c
curl -O https://people.engr.ncsu.edu/dbsturgi/246/exercise/exercise13/hello.h
curl -O https://people.engr.ncsu.edu/dbsturgi/246/exercise/exercise13/libhello.so.1.0
```

You just copied a shared library named libhello.so.1.0, along with a header file (hello.h) that has a declaration for the one function defined in the library. The function is called getMessage(). It returns a dynamically allocated string containing a hello world message. This example is trivial, but it's typical of how a shared library might be distributed. The header file describes what's available in the library, so client code can be compiled to use the library. The library file (libhello.so.1.0) contains the compiled code. It's automatically loaded at start-up by client code that uses the library. You don't get the source code for the library; you don't actually need it to use the library.

You also get source code for a driver program named driver.c. It calls the getMessage() function provided by the library, prints out the message it gets, frees the memory for the message and then exits.

Preparing and Using the Library

On a Linux machine, shared libraries are normally installed in standard locations by an administrator. We're not going to do that on the EOS Linux machines (we can't), so we need to configure our environment so it will look for shared libraries in the current directory. Using the following command to set the LD_LIBRARY_PATH environment variable should do the this for you. You only need to do this once, but if you log out and log back in to complete the exercise later, you'll need to do it again:

```
$ export LD_LIBRARY_PATH=.
```

If you look at the name of the library, you see some versioning information at the end. This is used to permit a system find the most recent, compatible version of a library. As long as as the number right after the .so (the .1 for this library) is the same, the library is considered compatible. The next value (the .0 for this library) is called the minor number. It's used to distinguish newer, compatible releases of the library from older ones.

Normally, a shared library will be configured so it shows up under a few different filenames with less specific versioning information. This is done with symbolic links. These are like pointers in the file system. When you try to open a symbolic link, the operating system notices that it's a symbolic link and opens the file it points to instead. Run the following commands to create two symbolic links for the library. The first link says that the file, libhello.so.1.0 is the latest release of version 1 of the library. So, if you want to load libhello.so.1, then the file libhello.so.1.0 is the one you should use. The next link is used at compile time. It says, if you're compiling a program that uses this library, and you don't care what version you're using, then you'll get version ibhello.so.1.0.

```
$ ln -sf libhello.so.1.0 libhello.so.1
$ ln -sf libhello.so.1.0 libhello.so
```

Now, use ls to see what you've done. You should see something like the following. The two new files we just created are really symbolic links, pointing to the file that contains the current version of the library:

```
$ ls -l
total 11
-rw------- 1 dbsturgi ncsu  278 Mar 17 14:27 driver.c
-rw------- 1 dbsturgi ncsu  121 Mar 17 14:27 hello.h
lrwxr-xr-x 1 dbsturgi ncsu   15 Mar 17 14:51 libhello.so -> libhello.so.1.0
lrwxr-xr-x 1 dbsturgi ncsu   15 Mar 17 14:51 libhello.so.1 -> libhello.so.1.0
-rwx------ 1 dbsturgi ncsu 6198 Mar 17 14:27 libhello.so.1.0
```

Now, we should be able to compile and run our driver program. The following command compiles it. You should already recognize most of these gcc options. The -L. at the end says to look in the current directory for libraries, and the -lhello (that's a lower-case L) says to link with a library named libhello.so. That's where one of the symbolic links we created above gets used. If you delete the link named libhello.so, this compile would fail.

```
$ gcc -Wall -std=c99 driver.c -o driver -L. -lhello
```

Now, before you run this program, let's use ldd to take a look at the shared libraries it depends on. Running ldd on the executable should give you a report like the following. You can see our program will try to load version 1 of libhello.so when it's started. That's where the other symbolic link we made is going to be used. If we deleted the libhello.so.1 link, our program wouldn't be able to find the library it needed at startup.

```
$ ldd ./driver
      linux-vdso.so.1 =>  (0x00007ffeb2bcc000)
```

```
libhello.so.1 => ./libhello.so.1 (0x00007f9d8d517000)
libc.so.6 => /lib64/libc.so.6 (0x0000003918400000)
/lib64/ld-linux-x86-64.so.2 (0x0000003918000000)
```

Now, finally, let's try running the program. You should get the standard "Hello World." message. That's the string returned by getMessage() for the version of the library I'm giving you:

```
$ ./driver
Hello World.
```

Updating the Library

Now, let's replace the library with a newer version. We should be able to get the driver program to use the newer version of the library without re-compiling the driver. That's the point; without a re-compile, client programs should be able to automatically use updated, dynamically linked libraries, as long as they are compatible with the version they were compiled with.

Write your own version of the library. Create a file named hello.c, include the hello.h header file and define your own getMessage() function. Your function will need to malloc() space for the message you're going to return, fill in the message with a null-terminated string and then, well, return it. For your version of the library, return the string "Hello unity-id.", using your own unity ID. So, for example, if I was doing this exercise, my function would return "Hello dbsturgi."

To compile your version of the library, you'll need to give the -fPIC option on the gcc line. This tells the compiler to write position-independent code, so the code for the library doesn't have to occupy the same range of logical addresses for every program that's using the library.

```
$ gcc -Wall -fPIC -c hello.c
```

The command above created an object file for your library code. You need to get the linker to make a shared library from this object. Typically, this is where you'd combine lots of library files into a single library, but the shared library we're making is trivial, containing just one object with just one function. In the following command, we're telling gcc to build a shared library and write it to an output file named libhello.so.1.1 (so, a newer release of the same library). The -Wl option and the syntax immediately afterward (that's a lower-case L) tells the linker that we're building a library that should be compatible with version libhello.so.1.

```
gcc -shared -Wl,-soname,libhello.so.1 -o libhello.so.1.1 hello.o
```

To get our driver to use the newer version of the library, we need to update the symbolic links, so they point to the new release of the library. Really, you just need to replace the first of these links, but we might as well keep them both consistent:

```
$ ln -sf libhello.so.1.1 libhello.so.1
$ ln -sf libhello.so.1.1 libhello.so
```

Now, try running the driver program again. You haven't recompiled it (hopefully), but it should automatically get the newer library when it starts up. Here's what I get:

```
$ ./driver
Hello dbsturgi.
```

For this exercise, just submit your shared library, libhello.so.1.1 to the exercise_13 assignment. Usually, you're asked to submit source files rather than binaries, but, in this case, submitting the library will help to show that you did the exercise. I'll grade it by linking with a different driver program and seeing if it works. I'll also be looking for your unity ID, so be sure you get this part right.