Lab 10 – Creating static and shared libraries (Week of 11.4.16)

Checkoffs for this lab are due within the lab period. To get checked off on this lab, you need to demonstrate your work to a TA.

Making a Static Library

First, we'll make a static library and link to it.

Step #1: Write code for a library

We'll make a library containing two files so you can see how to link together multiple object files into the same library.

Make one file called **`libfile1.c`** that contains the following code:

```
#include <stdio.h>

void lib_func1 ()
{
        printf("lib_func1() called.\n");
}
```

Now, make another file called **`libfile2.c`** that contains the following code:

```
#include <stdio.h>

void lib_func2 ()
{
        printf("lib_func2() called.\n");
}
```

Lastly, make a header file called **`libtest.h`** that looks like this:

```
void lib_func1();
void lib_func2();
```

Strictly speaking, we don't actually need to make a header file, we could use the library without it. The advantage to making a header file is that you get extra compile-time type checking on the functions you call, which is a good way to avoid subtle and confusing bugs.

Step #2: Write a Makefile that can compile the library

There are basically two steps in creating any shared library:

1. Compile the source (`.c`) files into object files with `gcc -c sourcefile.c`. The `-c` flag means "compile" without linking, thusly creating a `.o` (object) file.
2. *Archive* all the object files together by running the `ar` program.

Here is an example Makefile that will do the trick:

```
CC=gcc -g

# Build the static test library
libtest.a: libfile1.o libfile2.o
	ar rcs libtest.a libfile1.o libfile2.o

# Make an object file for the first library source file
libfile1.o: libfile1.c
	$(CC) -c libfile1.c

# Make an object file for the second library source file
libfile2.o: libfile2.c
	$(CC) -c libfile2.c
```

Note that the steps listed above are performed in reverse order in the Makefile. This is because *top-level* targets are usually (almost always) listed before dependent targets.

Note also the use of the Makefile variable `CC=gcc -g` and the use of it later on as `$(CC)`.

The `ar` program is used to make static libraries. Briefly, c = create, r = replace existing files, and s = write object file symbols. These are pretty much the only flags you need.

Build your library by typing **`make`** at the command line.

You can now examine the contents of your library by typing **`objdump -a libtest.a`**. If you want to see it interspersed with source code (both C and assembly) you can type `objdump -aS libtest.a`. (We can see the source code because we compiled with the -g flag, which generates debugging symbols.)

Step #3: Write a program that uses the library you just wrote

Create a little program called **`static_test.c`** that contains the following code:

```
#include "libtest.h"

int main (void)
{
        printf("static_test is about to call statically-linked "
                        "functions in libtest.a...\n");

        lib_func1();
        lib_func2();
}
```

Note that we are #include-ing the header file for our library (libtest.h). That's all there is to it.

Step #4: Modify the Makefile to compile `static_test.c`

To compile both the `static_test.c` program you just wrote as well as the library you wrote earlier, you can modify your Makefile so that it looks like this:

```
CC=gcc -g

# Build the staticly-linked test program
static_test: static_test.c libtest.a
        $(CC) static_test.c -o static_test -L./ -ltest

# Build the static test library
libtest.a: libfile1.o libfile2.o
        ar rcs libtest.a libfile1.o libfile2.o

libfile1.o: libfile1.c
        $(CC) -c libfile1.c

libfile2.o: libfile2.c
        $(CC) -c libfile2.c

clean:
        rm -f *.o *.a *.so static_test
```

Note that we have made the `static_test:` target dependent on both `static_test.c` (the source file for the program) and `libtest.a` (the library used by the program) so that it will compile the library if it needs to.

The `-L` flag passed to the compiler is used to add a directory to the list of directories that the compiler will traverse when it searches for libraries. Saying `-L./` means "add the current directory to the list of search dirs". The default locations that are searched for libraries are found in the file `/etc/ld.so.conf`. (Basically any path with a directory called "lib".) Because the library we wrote is not in a location that is automatically searched, we cannot link to it unless we explicitly tell the compiler where to find it.

The `-l` (lower case ell) option is used to specify the name of a library. Note that you do not say `-llibtest`, but just `-ltest`. The lib- prefix is assumed; this is standard.

Note also that we have added a "clean" target so we can just type `make clean` to clean up all our intermediate files. Making a "clean" target is a fairly common practice.

Now you can type **`make`** to build everything. You can run your program by typing **`./static_test`**. You should see the following output:

```
static_test is about to call statically-linked functions in
libtest.a...
lib_func1() called.
lib_func2() called.
```

Making a Dynamic (Shared) Library

Now let's look at making a shared library. There are several differences between a static and shared library. The main difference is that when you link to a static library, all the necessary code from the static library is imported directly into the calling program. In contrast, when you link to a shared library, no code is imported, the calling program simply calls it from the shared library. We'll get a hands-on look at this difference at the end of this page.

Step #5: Edit your Makefile to build a dynamic library

We don't have to write any new code for a shared library, we can just add a new target to our Makefile to build a shared lib in addition to a static lib. The following lines accomplish this:

```
# Build the dynamic test library
libtest.so: libfile1.o libfile2.o
        $(CC) -shared -o libtest.so libfile1.o libfile2.o -Wl,-
soname,libtest.so
```

The `-shared` flag tells the compiler we're making (you guessed it) a shared library.

The `-Wl` flag is used to pass arguments to the linker. Some compilers make you call the compiler first (usually a program called "cc") and the linker second (usually a program called "ld") but gcc allows us to call both programs and pass arguments to both on just one line. To distinguish compiler arguments from linker arguments, a comma is used to separate them instead of spaces. Thus, we have `-soname` as an option passed to the linker and `libtest.so` as an option argument, which tells the linker what the name of this library will be.

Step #6: Write a program that uses your shared library

We actually could use the code from our "static_test.c" program, but we would like to make a second program to show an important distinction later on. So, please put the following code in a file called **`shared_test.c`**.

```
#include "libtest.h"

int main (void)
{
        printf("shared_test is about to call dynamically-linked "
                        "functions in libtest.so...\n");

        lib_func1();
        lib_func2();
}
```

Step #7: Modify the Makefile to compile `shared_test.c`

This can be done by adding the following target:

```
# Build the dynamically-linked test program
shared_test: shared_test.c libtest.so
        $(CC) shared_test.c libtest.so -o shared_test
```

There is nothing important about the order of the filenames. We could have just as easily wrote: `$(CC) libtest.so -o shared_test shared_test.c`. While we're at it, lets make an "all" target that will compile all the stuff we've written:

```
# Make everything
all: static_test shared_test
```

Be sure to put this target at the top of the file before all the others.

You can now compile everything just by typing **`make`** at the command line.

Step #8: Try to run `shared_test`

You will notice that if you type **`./shared_test`** at the command line you will get the following error:

```
./shared_test: error while loading shared libraries: libtest.so: cannot
open shared object file: No such file or directory
```

The reason why you are getting this error is that the program doesn't know where to look to find the libtest.so file. We can use the `ldd` program to illustrate this. Type **`ldd shared_test`** at the command line and you will get the following output:

```
libtest.so => not found
libc.so.6 => /lib/libc.so.6 (0x4001e000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Note that our libtest.so file was not found? Why not, you ask? To answer this question, you need to know a little bit more about how the operating system works.

When the operating system runs programs that use shared libraries, it looks in a set of standard locations to find them: `/lib`, `/usr/lib`, and (often) `/usr/local/lib`. (Type an 'ls' in any of those directories just for kicks.) The libtest.so that we just made is installed in a *non-standard* location, so we need to give the operating system a special *run-time* directive to tell it where to find the library. This can be done simply by typing:

```
export LD_LIBRARY_PATH=`pwd`
```

The LD_LIBRARY_PATH is an environment variable that tells the linker where to look at run-time for libraries that are installed in non-standard locations. The `pwd` means "print working directory" and putting backtics around it (that's the one in the upper-left corner of your keyboard next to the TAB or the '1' key) means "use the output from this command, not the string itself". (Do not use the single quote by the RETURN key.) You can view the contents of this (or any other environment variable) by typing **`echo $LD_LIBRARY_PATH`**.

Now when you type `ldd shared_test` it is able to find the shared library. Furthermore, when you run the program **`./shared_test`** it should run like so:

```
shared_test is about to call dynamically-linked functions in
libtest.so...
lib_func1() called.
lib_func2() called.
```

(Alternatively, you can set the environment variable and run the program all on one line by typing ``LD_LIBRARY_PATH=`pwd` ./shared_test``.)

Step #9: Note the differences

At this point, you can see the difference between linking to a static or a shared library by using the `nm` program, which lists symbols in object files.

First, type `nm static_test` and you will see it produce numerous lines showing the various elements of the executable with a one-letter description of each. Anything with an upper-case letter 'T' is a function in the executable (other symbols are described in the man page for nm). Among other things, `nm` lists the following functions:

```
08048480 T _fini
08048298 T _init
08048310 T _start
08048414 T lib_func1
0804842c T lib_func2
080483f0 T main
```

(You can list just the functions with `nm static_test | grep " T "`.) Note that the two static library functions were imported into the executable. The 'main' function is part of every C program, and the ones with leading underscores were added by the compiler (boilerplate stuff).

Now try typing `nm shared_test`. You should get the following lines in your output:

```
        ...
```

```
080485a4 t init_dummy
         U lib_func1
         U lib_func2
08048550 T main
        ...
```

Note that `lib_func1` and `lib_func2` show up as "undefined" ('U') and with no addresses. This is because a stub for these functions exists in the executable, but the actual code for the function does not; it lives in the shared library instead.