Shared Objects

by John Fehr

Shared objects (SO) can be your best friend, or your worst enemy, depending on how you use them. If you use them simply as shared libraries linked with your application, everything is fine. If your shared libraries assume that certain symbols will be found at load time, your application will be searched for the missing symbols and the holes will be plugged. You can even override functions in the shared library if you define them in your application.

How about if you need to load the SO dynamically using **dlopen**()? Unfortunately, the linker doesn't resolve symbols from a **dlopen**()ed shared library using your application's symbols. This is because the symbols aren't in its dynamic symbol table. All of the symbols in the SO have to be resolvable without using your application's code. So what do you do if your shared library needs to call functions in your application?

Solution 1: Linker options

If you link your app with the --export-dynamic (or **-E**) option, the linker will place all of the application's symbols in the dynamic symbol table. This will make them available to your **dlopen**()ed shared library.

There are a few problems with this. The first is that if you strip your app, it'll most likely strip those symbols away. The second is that all of your symbols will be put in the dynamic symbol table. You might not want to give the shared object access to all of your symbols. Finally, exporting all of your symbols may result in a large increase in application size (which is a bad thing).

Solution 2: Function pointers

One solution is to have a set of global function pointers in the shared library that you set to the application's corresponding functions. This can be done either by doing a bunch of **dlsym**()s, searching for the function pointers in the library, and then setting them. Or it can be done by putting your application's functions in a structure of function pointers and then calling a special function in your shared library that copies the appropriate pointers where they're needed.

This solution works for the most part, but it also has a few problems. First, calling functions via function pointers is much slower than calling functions directly, since the CPU won't be able to do branch prediction. Second, it's hard to make changes to the structure of function pointers if you need to later, without breaking code. Also, if you decide to link the library later as a shared library (instead of opening with **dlopen**), you'll have to make a lot of changes to both the shared object and the application.

Is there a better solution?

Solution 3: Fooling the linker

Let's think again how the runtime linker works. Let's say you have a library, *liba.so*, that contains the function **a**(). In your application you call the **a**() function and link with *liba.so*. When you run the application, the **a**() function in *liba.so* gets called, just as you'd expect.

```
---------------------- Makefile ----------------------------
all: liba.so testso
liba.so: liba.c
 qcc -shared -fPIC liba.c -o liba.so
```

```
testso: testso.c liba.so
 qcc testso.c -o testso -L. -la
------------------------------------------------------------
---------------------- liba.c ------------------------------
#include
void a(void)
{
 printf("a() in liba.son");
}
------------------------------------------------------------
--------------------- testso.c ---------------------------
#include
extern void a(void);
main()
{
 a();
}
------------------------------------------------------------
```

If you try this out, you'll see '**a**() in *liba.so*' printed to your terminal. Seems pretty normal? Now, write your own **a**() function in *testso.c*:

```
---------------------- testso.c ---------------------------
#include
void a(void)
{
 printf("a() in testso.cn");
}
main()
{
 a();
}
------------------------------------------------------------
```

If you've read my “Doing Your Best without Crashing the Rest” article, you won't be surprised that there are no linking problems, and when run, it prints '**a**() in *testso.c*' to your terminal. How can this behavior help you with your **dlopen**()ed problem?

More on the linker

When you **dlopen**() a shared object, the linker needs to be able to resolve all the symbols in the object. If there are any undefined symbols, it tries to find them in any other shared libraries that are already loaded into your application's memory space. If it's unable to find them there the **dlopen**() fails.

However, when any shared library is loaded into memory at run time, any symbols that are common to the shared library and the application are replaced inside the shared library. (The shared library and the application will both use the application's symbols.)

The shared library is linked (and symbols are resolved/replaced) before the **dlopen**() for your shared object is called. And, since the linker resolves unresolved symbols in the **dlopen**()ed shared object against symbols in any loaded shared library, your net result is that the unresolved symbols in the **dlopen**()ed shared library resolve against your application's functions!

In English, please?

This sounds a bit confusing, perhaps. Let's walk through exactly what you want to happen:

- You have a shared object, *coolobj.so*, that has a function **b**() that needs to call a function **a**() in whatever application it's loaded into.
- You have your application, testso, that contains a function **a**() and needs to call a function **b**() in a **dlopen**()ed shared object. Function **b**() needs to call the application's **a**() function.
- You have a shared library, *liba.so*, that contains a dummy function **a**().
- You create testso and link it with liba.so.
- You run testso. Your testso application loads *liba.so*, but since testso contains the **a**() symbol it replaces the pointer for symbol **a**() in *liba.so* with the **a**() function pointer in *testso*.
- After it's loaded into memory, testso **dlopen**()s *coolobj.so*. The linker sees that **a**() isn't defined internally in *coolobj.so* so it attempts to find it in any other shared libraries that have already been loaded. It will, of course, find the **a**() symbol in *liba.so* which has been replaced by **a**() in testso and it will resolve to that **a**() symbol.
- When **b**() in *coolobj.so* is called, it will call your application's **a**() function.

Here's a new Makefile, *testso.c*, and *coolobj.c*:

```
---------------------- Makefile ----------------------------
all: liba.so testso coolobj.so
liba.so: liba.c
 qcc -shared -fPIC liba.c -o liba.so
testso: testso.c liba.so
 qcc testso.c -o testso -L. -la
coolobj.so: coolobj.c
 qcc -shared -fPIC coolobj.c -o coolobj.so
------------------------------------------------------------
---------------------- testso.c ----------------------------
#include
#include
void a(void)
{
 printf("a() in testso.cn");
}
main()
{
 void *obj=dlopen("coolobj.so",RTLD_GLOBAL);
 void (*b)(void);
 b=dlsym(obj,"b");
 b();
 dlclose(obj);
}
------------------------------------------------------------
---------------------- coolobj.c ---------------------------
#include
extern void a(void);
void b(void)
{
 printf("b() in coolobj.son");
 a();
```

```
}
-------------------------------------------------------------
```

If you build and try out testso, you'll get:

```
b() in coolobj.so
a() in testso.c
```

Which is exactly the behavior you want!

On second thought...What if you decide later that you want to use *coolobj.so* as a real shared library after all and link against it at compile time?

All you'd have to do is get rid of the **dl*** function calls and simply call the extern function **b**() instead. No changes would have to be made in your *coolobj.so* source code at all!

Of course, you might find it easier to first develop *coolobj.so* as a normal shared library and later convert it to a **dlopen**()ed shared object. Again, you won't need to make any changes to *coolobj.so*, just to the application.

So is it faster?

I wrote two simple little sample programs/shared objects that illustrate the speed difference between using our little dummy shared library and setting/using function pointers which you can download at: http://www.altbits.com/jfehr/art/dummylib.zip

I've also included a version that doesn't use **dlopen** at all, but simply links with the shared object. The speed difference isn't astounding (about 10%), but it clearly shows which method is faster. Even more surprising, at least on my machine, the dummy shared library method is 5% faster then the method that doesn't use **dlopen** at all!

Best of all, you've overcome the problems of solutions 1 and 2:

- Stripping won't make your application stop working with your shared object.
- You only give the shared object access to the functions in the app that you want.
- The added size of the dummy shared library isn't nearly as great as the added size of the application would have been if we had linked our application with the **-E** option instead of using the dummy shared library.
- There's no function pointer passing and no slowdown due to branch prediction.
- You don't have to worry about structures changing.
- It's easy to link as a shared library if you decide you don't want the shared object **dlopen**()ed, but instead linked to your application.

You be the judge!