Doing Your Best Without Crashing the Rest

by John Fehr

So you've written this amazing shared library that you're using in many of your critical services and apps. One day, you discover that you could have written one of the internal functions in the shared library much more efficiently. (This could be in terms of network usage, memory usage, disk usage, etc.) However, since the shared library is now being used by critical components of your system, you can't just replace it with your new and improved shared library. If you've made an error somewhere in your new code, it will affect all those components which at best will render your system unusable. So what's the best way to test how well this new code gets along with the rest of the shared library?

The answer is actually quite simple. By default, shared libraries resolve their symbols at load time, not at link time. So when the shared library uses a function, say **foo**(), it first searches the application's code for this function, and then its own. All we have to do is define this function in our own code, and it will be used instead of the shared library's function, even internally within the shared library!

Here's an example:

Let's say you have a shared library called *libcoolstuff.so*. In this library, there's a function called **copystrings**(), which copies a given number of strings of a maximum length of 100 from one location to another. This **copystrings**() function also uses another function, **copystring**(), which copies a single string from one location to another. The second function also exists inside the **coolstuff** library, but is generally not called from outside the **coolstuff** library. Here's our *coolstuff.c* file:

```
void copystring(char *src,char *dst)
{
 strcpy(src,dst);
 sleep(1);
}
void copystrings(char src[][100],char dst[][100],int nstrings)
{
 int i;
 for (i=0;i<nstrings;i++)
  copystring(src[i],dst[i]);
}
```

Now consider an application that uses the **coolstuff** library. We'll call it *coolapp.c*:

```
#include <stdio.h>
void copystrings(char src[][100],char dst[][100],int nstrings);
main()
{
 char src[32][100];
 char dst[32][100];
 int i;
 for (i=0;i<32;i++) sprintf(src[i],"String %d",i);
 copystrings(src,dst,32);
 for (i=0;i<32;i++)
  if (strcmp(src[i],dst[i])!=0)
   printf("src[%d]='%s', dst[%d]='%s'n",i,src[i],i,dst[i]);
}
```

Here's a makefile you can use to compile this, so you can test it:

```
all: libcoolstuff.so coolapp

libcoolstuff.so: coolstuff.o
 $(CC) -shared coolstuff.o -o libcoolstuff.so

coolapp: coolapp.o libcoolstuff.so
 $(CC) -o coolapp coolapp.o -L. -lcoolstuff

%.o: %.c
 $(CC) -c $< -fpic -o $@
```

Type make at the command line, and run **coolapp**. You'll notice that it takes about 32 seconds to finish. There is a way to speed this up, but we don't want to cause other applications using this shared library to crash, particularly if we have a bug in our new code. So, let's add our experimental **copystring**() function in the *coolapp.c* file! Try this:

```
void copystring(char *src,char *dst)
{
 strcpy(src,dst+1);
}
```

Now re-make, and run coolapp again. Luckily, this version of **copystring**() isn't in our shared library, or our other applications that are using it wouldn't have worked correctly! Let's change the function to:

```
void copystring(char *src,char *dst)
{
 strcpy(src,dst);
}
```

Again, re-**make**, and run **coolapp**. This time, it finishes much faster, and doesn't report any errors.

Such behavior can, however, lead to some unpredictable and unexpected results if you don't already know that certain functions exist in a shared library you are using. For example, let's say you were using the **coolstuff** shared library, but you only knew about the **copystrings** function. You decide that your program needs a function to copy a string contained after the first ***n*** in another string, and you call it **copystring**:

```
#include <stdio.h>
void copystring(char *src,char *dst)
{
 char *start=strchr(src,'n');
 if (start)
  strcpy(dst,start+1);
 else
  strcpy(dst,"");
}
main()
{
 char src[32][100];
 char dst[32][100];
 char second[100];
 int i;
 for (i=0;i<32;i++) sprintf(src[i],"String %dnSecond line.",i);
 copystrings(src,dst,32);
```

```
  for (i=0;i<32;i++)
   if (strcmp(src[i],dst[i])!=0)
    printf("src[%d]='%s', dst[%d]='%s'n",i,src[i],i,dst[i]);
  copystring(src[0],second);
  printf("second string of '%s' is '%s'n",src[0],second);
 }
```

If you knew nothing about the existing **copystring**() function in our shared library, you would most likely be quite puzzled as to what was happening. In this example, it wouldn't be too difficult to figure it out, but in a real (useful) shared library and application, things are usually not this straightforward.

There are four possible solutions:

1) If you have the original source code to the shared library, rebuild it as a static library, and link against that instead. That way, it will tell you about any duplicate symbols, and you can change your function name so it no longer matches.
2) If you don't want this kind of behavior (where the application can override the internal functions of your shared library), then link your application with the **-Bsymbolic** option. (Or **-Wl,-Bsymbolic** if you compile using qcc.) That option tells the linker to resolve all your symbols internally at link time instead of at runtime. In this case, all of your symbols will have to be defined in the object files and shared libraries you specify on the linker command. (For example, if your shared library uses Photon, you'll have to have **-lph** on your linker command, so it knows where to resolve Photon functions.)
3) Make your internal functions static. Of course, that also means the static function has to be in the same source file as the functions that use it. This is probably a better solution for those internal functions that aren't used in many different source files.
4) Make your internal function names so random and confusing that even you can't figure out what functions do what.

A good idea might be to build and release a shared library with option #2 for your customers, and build another one without option #2 for your own internal testing and modification. That way, your customers won't accidentally shoot themselves in the foot, and you can still test new code without harming your other running applications.

Happy and safe coding!