CS 0449 – Shared Object Lab

Background

A *Shared Object* is an executable file that acts as a shared library of functions. Dynamic linking provides a way for a process to call a function that is not part of its executable code. The executable code for the function is located in a .so file, which contains one or more functions that are compiled, linked, and stored separately from the processes that use them. Shared objects also facilitate the sharing of data and resources. Multiple applications can simultaneously access the contents of a single copy of a shared object in memory.

Dynamic linking differs from static linking in that it allows an executable file to include only the information needed at run time to locate the executable code for a shared object function. In static linking, the linker gets all of the referenced functions from the static link library and places it with your code into your executable.

Procedure

1. Login via SSH to thot.cs.pitt.edu

2. Change directories to the one we created for the work for this class

3. The first step is to make a library. Let us imagine we are writing our own string library. One function that we might want to provide is string copy. Edit a file in your favorite text editor, or pico like:

```
pico mystr.c
```

4. Now enter the code necessary to make a string copy function:

```
void my_strcpy(char *dest, char *src)
{
	while(*dest++ = *src++);
}
```

5. Save the file

6. We now need to compile this into a library. The first step is to invoke the compiler independently of the linker. We can do this by providing the -c option to gcc. Also, since we do not know the address in memory where the shared object will be loaded, we need to tell gcc to create position-independent code via the -fPIC option:

```
gcc -fPIC -c mystr.c
```

7. This creates a mystr.o object file. We can now invoke the linker separately, telling it to create a shared object file with the -shared flag. The -soname option allows us to specify the name of the output file, as well as the version number, which is placed after the .so extension. Finally, -lc tells the linker to also link in the functions from libc.

```
ld -shared -soname mystr.so.1 -o mystr.so.1.0 -lc mystr.o
```

8. We want this to be accessible via mystr.so without any version number, so we need to make a *symbolic link* to the file with a new name. A symbolic name is much like a shortcut in Windows.

```
ln -s mystr.so.1.0 mystr.so
```

9. Finally, we normally want to install libraries to the standard place on a UNIX or Linux system (/lib or /usr/lib) but we cannot do that on thot. Instead, we can tell Linux that this directory is an appropriate place to look for libraries by issuing the command:

```
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

10. We now need to write a program to use our new function. The function to open a shared object is dlopen. Read the man page to learn a bit more about it:

```
man dlopen
```

11. Make a file named main.c with the following contents:

```c
#include <stdio.h>
#include <dlfcn.h>

int main() {
   void *handle;
   void (*my_str_copy)(char *, char *);
   char *error;
   handle = dlopen("mystr.so", RTLD_LAZY);
   if(!handle) { //handle == NULL
      printf("%s\n", dlerror()); //dlerror gives us a string with the error
      exit(1);
   }
   dlerror();    // Clear any existing error
   my_str_copy = dlsym(handle, "my_strcpy"); //lookup the function by name
   if ((error = dlerror()) != NULL)  {
      printf ("%s\n", error);
      exit(1);
   }
   //Let's test it.
   char dest[100];
   char src[] = "Hello World!";

   my_str_copy(dest, src);
```

```
    printf ("%s\n", dest);
    dlclose(handle);
    return 0;
}
```

12. Notice that we use the dlopen function to load the library and then ask for a function pointer to a particularly named function via dlsym. The RTLD_LAZY makes the loading truly use Dynamic Loading as we have discussed in class. The other thing to notice is the declaration of the function pointer. It points to a function that returns void, and takes two char * parameters, just as our declaration of my_strcpy did.

13. We now need to compile the file. We tell gcc to link the code via dynamic linking (-rdynamic) and to include in our executable libdl which contains the implementation of dlopen and friends:

```
gcc -rdynamic -o main main.c –ldl
```

14. We can now run the program and see the intended results:

```
./main
Hello World!
```

Your turn

Add an implementation of my_strcat to mystr.c and test it in main.c You do not need to do steps 8, 9, or 10 again (unless you've logged out and then back in, in which case you'll need to redo step 9).

Show the TA when you are done.