CS220 - Computer System II
Lab 5

Due: 09/28/2016, 11:59pm

1 Introduction

Hopefully you are now comfortable with writing programs in C. Write-once-and-reuse-many-times is a valuable practice in programming. Code libraries are essential for reusing code. Understanding how libraries work is a key component of C software development. In this lab, you will learn about two main forms of reusing binary code, shared and static libraries. You will implement a library that allocates and deallocates memory on the heap. It will print log statements to record all the allocations and deallocations. You will write a program to test the shared and static library versions of the library.

2 Getting Started

Create a directory called Lab5 in your home directory or any other convenient location. You will implement all the code here.

3 Implementing the library

Within Lab5, create `shared.h` with the following contents.

```
/* shared.h */
#ifndef _SHARED_H
#define _SHARED_H

extern void * logger_malloc (unsigned int size);
extern void logger_free (void *p);

#endif

```

The `extern` keyword allows one module of your program to access a global variable or function declared in another module. It tells the compiler that the definition is provided somewhere else.

Next, create `shared.c` with the following contents.

```
/* shared.c */
#include <stdio.h>
```

```c
#include <stdlib.h>

void *logger_malloc(unsigned int size)
{
  void *ret;
  printf("Allocating %u bytes...\n", size);
  ret = malloc(size);
  if(ret == 0) {
    printf("Allocation failed! :(\n");
  } else {
    printf("Successfully allocated at %p\n", ret);
  }
  return ret;
}

void logger_free(void *p)
{
  printf("free()ing memory at %p...", p);
  free(p);
  printf("DONE\n");
}
```

The functions `logger_malloc` and `logger_free` are wrappers for malloc and free that simply print log statements.

4 Static library

While using static library, required dependencies are statically linked to the binary during compilation. In other words, the required functions are embedded into the target binary. Static libraries typically have a `.a` extension.

Compile shared.c using the following command:

```
$ gcc -c -std=c89 -g -Wall shared.c -I$PWD
```

The `-c` option tells gcc to simply compile the program and generate an object file shared.o, but not generate an executable (an executable can execute, and must contain a main function). In shared.c, note that shared.h is included as `#include<shared.h>` as

opposed to `#include "shared.h"`. Files, stdio.h, stdlib.h, etc. are already included in the default search paths. The -I option tells gcc additional directories to look for include files in. Therefore, -I$PWD tells gcc to look for shared.h in the current directory.

Next, use the following archiver (ar) command to generate an archive of the object file(s) to generate the `.a` file. The archiver is used to package multiple object files into a single file:

```
ar rcs libshared.a shared.o
```

▷ Look up the manpage for `ar` to see what options `rcs` do!

Make a directory called `lib` and move libshared.a to lib:

```
$ mkdir -p lib
$ mv libshared.a ./lib/libshared.a
```

5 Shared library

Shared Libraries are the libraries that can be linked to any program at run-time. They provide a means to use code that can be loaded anywhere in the memory. Once loaded, the shared library code can be used by any number of programs. libc is an excellent example of shared library.

Use the following commands to compile shared.c and generate the shared library:

```
$ gcc -c -std=c89 -g -Wall shared.c -I$PWD
$ gcc -shared -o libshared.so shared.o
```

The `-shared` option tells gcc to create a shared library. `libshared.so` will be the name of the shared library. The shared library must start with `lib` and must have an extension `.so`. Move libshared.so to `lib` directory.

6 Using the libraries

Create `main.c` with the following contents:

```
#include <shared.h>

int main()
{
void *p;
p = logger_malloc(10 * sizeof(unsigned int));
logger_free(p);
return 0;
}
```

Statically linking libshared.a Use the following command to statically link libshared.a to main.c.

```
$ gcc -std=c89 -g main.c -I$PWD lib/libshared.a -o main_static
```

The library libshared.a is embedded into the resulting `main_static` executable! To test it, extract the disassembly of the main program:

```
$ objdump -d main_static > main_static.disas
```

Open it and search for logger_malloc and logger_free.

▷ Run main_static and record the output in Lab5.txt. Record the addresses of logger_malloc and logger_free functions in disassembly of main.

Dynamically linking libshared.so Dynamic linking is different when compared to static linking. The dependencies are resolved at runtime. Build main using the following command:

```
$ gcc main.c -I$PWD -L$PWD/lib -o main_shared -lshared
```

The -I option tells gcc where to find shared.h. Similarly, the -L option tells gcc where to look for the shared library. The -l option tells gcc what the dependent library is. Note that although the name is libshared.so, the -l option accepts "shared". The preceding "lib" and the extension are implicit. The information regarding what the dependency is (libshared.so) is encoded into the binary (main_shared), but not the library itself.

▷ Run main_shared and *record the error output* in Lab5.txt.

Now, run the following command and see that **the shared library location is unknown**:

```
$ ldd main_shared
```

Now, **retry running** main_shared after setting the LD_LIBRARY_PATH variable:

```
$ bash
$ export LD_LIBRARY_PATH=$PWD/lib
```

Shared libraries are loaded at runtime. Therefore, at runtime, the loader and dynamic linker (which is different from gcc) needs to know where to find the libraries. The variable LD_LIBRARY_PATH tells the loader where to look for shared libraries. The export command is not supported on the tee-shell, which is the default shell on remote.cs.binghamton.edu. So, we run bash before running export.

To test, run the following commands and record the outputs in Lib5.txt.

```
$ ldd main_shared
$ ldd main_static
```

The program ldd lists the shared library dependencies of programs.

7 Submitting the result

Remove binaries and intermediate files from Lab5. Create a tar.gz of Lab5 folder with only main.c, shared.h, shared.c and Lab5.txt.:

```
$ cd ..
$ tar -cvzf lab5_submission.tar.gz ./Lab5
```

Submit lab5_submission.tar.gz.