Lesson 12

Christian Schwarz, Jakob Krebs

27.1.2020

Contents

Libraries

- Static Libraries
- Dynamic Libraries

Libraries

Static Libraries

- Compilation is hard, and can take a long time. The compiler needs to do a lot of work to get from C code to an executable binary.
- When our Project becomes large, or if we are using a lot of existing code, we might not want to recompile everything every time we build.
- For example: We commonly use the C Standard Library in our programm, but we never actually change it.
- How can we avoid uneccessary recompilation?

Static Libraries

- Remember that compilation involved multiple steps:
  1. Preprocessing
  2. Compiling
  3. (Assembling)
  4. Linking
- Steps 1-3 run for every file. The output doesn't change, unless the C file (or the included headers) change.
- So should we just link all the .o / .obj files of libc and other dependencies every time? Doesn't that get annoying?
- GNU/Linux allows us to bundle multiple .o files into a bundle (think .zip) called an archive (.a)
- On Windows multiple .obj's can be bundled into a .LIB file
- We call both .a and .LIB static libraries, since they define a bunch of functions (and global variables) that we can use in our applications.

Creating a Static Library

```c
//hello.c
#include <stdio.h>
void hello_can_you_hear_me(void){
    puts("Hello from the other side!");
}
```

```
$ gcc -c hello.c
$ ar rcs libhello.a hello.o
```

- `-c` makes gcc stop after the compilation step and generate object files instead of an executable
- the ar utility (with the parameters rcs) creates our archive
- we could have appended multiple files for both programs
- prefixing libraries with `"lib"` is a typical linux convention

Using a Static Library

```c
//main.c
void hello_can_you_hear_me(void); //forward declaration
int main(){
    hello_can_you_hear_me();
    return 0;
}
```

```
$ gcc main.c test.a
$ ./a.out
Hello from the other side!
$
```

Dynamic Libraries

- In a normal Desktop Operating System, many processes run at the same time.
- For every Process, the OS loads the code inside the executable into main memory.
- If one executable is launched multiple times, the physical memory for the code can be shared.
- But if all Libraries were statically linked into the executables, commonly used libraries like `libc` would end up being stored in RAM many times, creating a lot of redundancy.

- Also, when the Code for a library gets updates (bugfixes, performance improvements, etc.), all executables (and libraries) depending on that library would need to be recompiled and redistributed to the users.
- And by the way, how do you create optional modules / plugins?

Dynamic Libraries

- Dynamic libraries are completely different from static libraries, in that they get loaded during runtime of the program.
- The Application specifies the name of the dnymic libraries its wants to use, and during runtime these symbols get loaded by searching for files with the specied names and searching the requested symbols in them.

```
1 #include <dlfcn.h>
2 int main(){
3     void *dl_handle = dlopen ("libm.so", RTLD_LAZY);
4     double (*cosine)(double);
5     cosine = dlsym(dl_handle, "cos");
6     printf ("%f\n", (*cosine)(2.0));
7     dlclose(handle);
8 }
```

Dynamic Libraries

```
$ gcc -shared test.c -o libtest.so
$ gcc main.c -L . -l test
$ export LD_LIBRARY_PATH=./:$LD_LIBRARY_PATH
$./a.out
Hello from the other side!
$
```

- `-shared` requests building a shared library
- `-L` adds a directory to search for libraries (at compile time)
- `-l` adds a library name to search for (also later used at runtime)
- we must omit the leading `"lib"` so our library will be found
- `LD_LIBRARY_PATH` is an environment variable for adding additional directories that are searched for dynamic libraries (ld is the dynamic linker)
- we must add `"."` so the current directory is searched

Dynamic Libraries Trivia

- another useful variable is `LD_PRELOAD`, which allows us to preload arbitrary libraries and therefore symbols for our executable (e.g. used by strace, can have security implications)
- Windows works a bit different here, you have to specify the functions which you want to export for the dynamic library by prepending `__declspec(dllexport)` to every function, and forward declare them them with `__declspec(dllimport)`
- this is usually done using a macro like `DLL_EXPORT` in header files which changes meaning based on pp directives (e.g. expands to nothing on Unix)
- Windows also requires you to link a static wrapper library to your application for all DLLs. This wrapper .LIB gets automatically created when the DLL is compiled
- On the upside, windows automatically searches for DLLS in the application's directory