What is Static and Dynamic Linked Programs (with Video)

Libraries are files containing the object files of various programs that are used to make compilation much faster. In C, there are two main types of libraries: *static* and *dynamic*. *Static libraries* are specifically called in the linking phase of compilation and tend to be both bigger and slower than dynamic libraries.

Dynamic libraries, on the other hand, do not require the code to be copied. They are not loaded automatically when a program starts, and are, instead, linked when the program is run. This causes them to be more efficient and smaller than static libraries. *Dynamic libraries* are also known as *shared libraries* because the code is shared by the programs that use it; each program, however, maintains its own stack and heap, keeping all running programs separate from one another.

Statically Linked Programs

In statically-linked programs, all code is contained in a single executable module. Static libraries are files that contain object files called modules or members. Library references are more efficient because the library procedures are statically linked into the program. Static linking increases the file size of your program, and it may increase the code size in memory if other applications, or other copies of your application, are running on the system. Static libraries are used when **stability** and **loadtime** are most desired. There are no dependencies required — if you have successfully compiled the library with no linking errors, it will work indefinitely. Static libraries are typically named with an .a extension. .a stands for archive.

Creating a Static Library using the archive command ar

gcc -c questions.c verifyResponse.c getARandom.c

ar rcs lib**ngdemo**.a getARandom.o questions.o verifyResponse.o

gcc main.c -o **ngb** -L. -l**ngdemo**

./ngb

The option in the ar command are:

r inserts files into the archive, replacing any existing members whose names matches that being added. New members are added at the end of the archive.

s creates and updates the map that cross-references symbols to the members in which they are defined

c creates the archives if it doesn't exist from files, suppressing the warning ar would putput if archive doesn't exist

Dynamically Linked Programs

The operating system provides facilities for creating and using dynamically linked shared libraries. With dynamic linking, external symbols referenced in user code and defined in a shared library are resolved by the loader at load time. When you compile a program that uses shared libraries, they are dynamically linked to your program by default.

The idea behind shared libraries is to have only one copy of commonly used routines and to maintain this common copy in a unique shared-library segment. These common routines can significantly reduce the size of executable programs, thereby saving disk space. The shared library code is not present in the executable image on disk, but is kept in a separate library file. Shared code is loaded into memory once in the shared library segment and shared by all processes that reference it. Dynamically linked libraries therefore reduce the amount of virtual storage used by your program, provided that several concurrently running applications (or copies of the same application) use the procedures provided in the shared library. They also reduce the amount of disk space required for your program provided that several different applications stored on a given system share a library. Other advantages of shared libraries are as follows:

- Load time might be reduced because the shared library code might already be in memory.
- Run-time performance can be enhanced because the operating system is less likely to page out shared library code that is being used by several applications, or copies of an application, rather than code that is only being used by a single application. As a result, fewer page faults occur.
- The routines are not statically bound to the application but are dynamically bound when the application is loaded. This permits applications to automatically inherit changes to the shared libraries, without recompiling or rebinding.

Disadvantages of dynamic linking include the following:

- From a performance viewpoint, there is "glue code" that is required in the executable program to access the shared segment. There is a performance cost in references to shared library routines of about eight machine cycles per reference. Programs that use shared libraries are usually slower than those that use statically-linked libraries.
- Dynamically linked programs are dependent on having a compatible library. If a library is changed (for example, a new compiler release may change a library), applications might have to be reworked to be made compatible with the new version of the library. If a library is removed from the system, programs using that library will no longer work

How Libraries are loaded by ldconfig with admin privileges

On GNU glibc-based systems, including all Linux systems, starting up an ELF binary executable automatically causes the program loader to be loaded and run. On Linux systems, this loader is named /lib/ld-linux.so.X (where X is a version number). This loader, in turn, finds and loads all other shared libraries used by the program.

The list of directories to be searched is stored in the file **/etc/ld.so.conf**.

Searching all of these directories at program start-up would be grossly inefficient, so a caching arrangement is actually used. The program **ldconfig** by default reads in the file /etc/ld.so.conf, sets up the appropriate symbolic links in the dynamic link directories (so they'll follow the standard conventions), and then writes a cache to /etc/ld.so.cache that's then used by other programs. This greatly speeds up access to libraries. The implication is that ldconfig must be run whenever a DLL is added, when a DLL is removed, or when the set of DLL directories changes; running ldconfig is often one of the steps performed by package managers when installing a library. On start-up, then, the dynamic loader actually uses the file /etc/ld.so.cache and then loads the libraries it needs.

Creating a Shared Library without admin privileges

Creating a shared library is easy. First, create the object files that will go into the shared library using the gcc -fPIC or -fpic flag. The -fPIC and -fpic options enable ``position independent code'' generation,
a requirement for shared libraries. We outline simple steps to create a shared library without having to have superuser permission.

The compilation generates object files (using -c), and includes the required -fPIC option:

```
gcc -fPIC -c questions.c
gcc -fPIC -c verifyResponse.c
gcc -fPIC -c getARandom.c
gcc -shared -o libngdemo.so getARandom.o verifyResponse.o questions.o
gcc -fPIC -c main.c
gcc -o ngb main.o ./libngdemo.so
```

Here is the video:

https://www.youtube.com/watch?v=jiShZsP1IHg (https://www.youtube.com/watch?v=jiShZsP1IHg)

(https://www.youtube.com/watch?v=jiShZsP1IHg)

Concept behind calling library C routines

This webpage is evolving , so it is not in any quiz, assignment or exams. THIS IS ENTIRELY FYI

You might have to wonder how did the executable finds printf or any other library functions when I didn't include the library code in my C language. I didn't include the library during compilation too.

So , if you have a code like this

printf ("Hello \n") ?

printf ("World \n") ?

Where did my system find the code for printf ? How is it finding it ? You are absolutely right and as a computer scientist, you should wonder.

The compiler doesn't know where the printf code exists because the library Code will be loaded at any random place. So, it is going to insert a FILL IN THE BLANK for the dynamic loader to fill the blank.

Consider, all the fill in the blanks created by the compiler will be a table called Global Address Table (GOT) for the dynamic linker (DL) to fill it up. But compiler maintains another table called Procedure linkage table (PLT) where all procedures you called are listed, one entry for each function. Along with the procedure name in the PLT , a pointer to the GOT table is maintained. Something like this (I am trivializing it here)

```
main ( )
{
 printf .....
 printf
```

scanf .....

}

PLT

printf Index 0 to GOT

scanf index 1 to GOT

GOT

0 LOAD DL and FILL THIS BLANK

1 LOAD DL and FILL THIS BLANK

As you can see , GOT indexes 0 and 1 are calling dynamic loader to fill the blanks. When your main function is executing at printf, it goes to PLT and finds index 0 at GOT. But at GOT index 0, dynamic loader is launched and kicks in.

Dynamic loader now does the finds the library you need. You can also find this by giving this command

ldd a.out <enter>

DL goes on the hunting trip to find the printf function and of course it's address you called in the C library. It replaces the address at GOT 0 with the actual address of printf and executes printf for you. But you thought you are doing all this work :)

Here is the video I made on the gist of the DL

https://www.youtube.com/watch?v=r4auCn-_axU (https://www.youtube.com/watch?v=r4auCn-_axU)

A real world example to better explain this :

You made a reservation at the hotel. On the day of the arrival, you go to the checkin counter. The guy at the counter gives you room key and number. You spend the next day and return to your hotel. Do you ask the guy for the room number or do you already know it ? This is how DL, acting like the guy at the counter. The first time you call printf, DL kicks in and gives you the address. Next time you call, you remember the address and execute printf directly without calling DL.

Once the DL is done finding the printf, it will update the GOT table like this

GOT

0 Here is the address of printf (next time, don't bother me :))

1 LOAD DL and FILL THIS BLANK

I will upload the actual assembly code and the video later <stay tuned >