developerWorks®

Anatomy of Linux dynamic libraries

Process and API

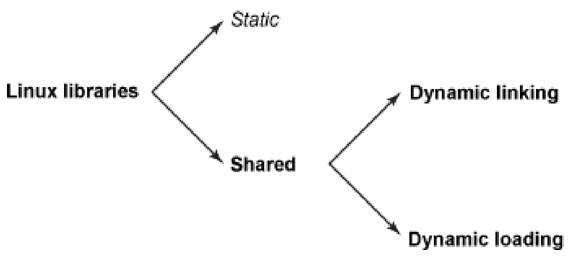
M. Tim Jones

August 20, 2008

Dynamically linked shared libraries are an important aspect of GNU/Linux®. They allow executables to dynamically access external functionality at run time and thereby reduce their overall memory footprint (by bringing functionality in when it's needed). This article investigates the process of creating and using dynamic libraries, provides details on the various tools for exploring them, and explores how these libraries work under the hood.

Libraries were designed to package similar functionality in a single unit. These units could then be shared with other developers and permitted what came to be called **modular programming**—that is, building programs from modules. Linux supports two types of libraries, each with its own advantages and disadvantages. The **static library** contains functionality that is bound to a program statically at compile time. This differs from **dynamic libraries,** which are loaded when an application is loaded and binding occurs at run time. Figure 1 shows the library hierarchy in Linux.

Figure 1. Library hierarchy in Linux

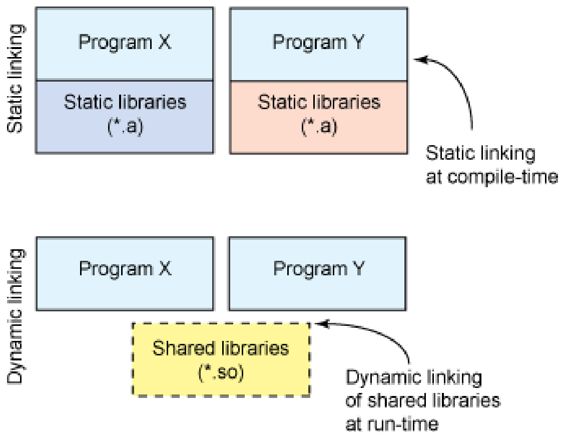


You can use shared libraries in a couple of ways: either linked dynamically at run time or dynamically loaded and used under program control. This article explores both of these methods.

Static libraries can be beneficial in small programs where minimal functionality is needed. For programs that require multiple libraries, shared libraries can reduce the memory footprint of the program (both on disk and in memory at run time). This is because multiple programs can use a shared library simultaneously; therefore, only one copy of the library is needed in memory at a time. With a static library, every running program has its own copy of the library.

GNU/Linux provides two ways to deal with shared libraries (each method originating from Sun Solaris). You can dynamically link your program with the shared library and have Linux load the library upon execution (unless it's already in memory). An alternative is for the program to selectively call functions with the library in a process called **dynamic loading.** With dynamic loading, a program can load a specific library (unless already loaded), and then call a particular function within that library. (Figure 2 shows these two methods.) This is a common usage pattern in building applications that support plugins. I explore this application program interface (API) and demonstrate it later in the article.

Figure 2. Static vs. dynamic linking



Dynamic linking with Linux

Now, let's dig into the process of using dynamically linked shared libraries in Linux. When users start an application, they're invoking an Executable and Linking Format (ELF) image. The kernel begins with the process of loading the ELF image into user space virtual memory. The kernel notices an ELF section called `.interp`, which indicates the dynamic linker to be used (/lib/ld-linux.so), shown in Listing 1. This is similar to the interpreter definition for script files in UNIX® (#!/bin/sh): It's just used in a different context.

Listing 1. Using readelf to show program headers

```
mtj@camus:~/dl$ readelf -l dl

Elf file type is EXEC (Executable file)
Entry point 0x8048618
There are 7 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x000e0 0x000e0 R E 0x4
  INTERP         0x000114 0x08048114 0x08048114 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x00958 0x00958 R E 0x1000
  LOAD           0x000958 0x08049958 0x08049958 0x00120 0x00128 RW  0x1000
  DYNAMIC        0x00096c 0x0804996c 0x0804996c 0x000d0 0x000d0 RW  0x4
  NOTE           0x000128 0x08048128 0x08048128 0x00020 0x00020 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4

  ...

mtj@camus:~dl$
```

Note that ld-linux.so is itself an ELF shared library, but it is statically compiled and has no shared library dependencies. When dynamic linking is needed, the kernel bootstraps the dynamic linker (ELF interpreter), which initializes itself, and then loads the specified shared objects (unless already loaded). It then performs the necessary relocations, including the shared objects that the target shared object uses. The `LD_LIBRARY_PATH` environment variable defines where to look for the available shared objects. When done, control is transferred back to the original program to begin its execution.

Relocation is handled through an indirection mechanism called the *Global Offset Table* (GOT) and the *Procedure Linkage Table* (PLT). These tables provide the addresses of external functions and data, which ld-linux.so loads during the relocation process. This means that the code that requires the indirection (that is, uses the tables) needs no changes: only the tables require adjustment. Relocation can occur immediately upon load or whenever a given function is needed. (See more on this difference later in Dynamic loading with Linux.)

When the relocations are complete, the dynamic linker allows any loaded shared object to execute optional initialization code. This functionality allows the library to initialize internal data and prepare for use. This code is defined in the `.init` section of the ELF image. When the library is unloaded, it may also call a termination function (defined as the `.fini` section in the image). When the initialization functions have been called, the dynamic linker relinquishes control to the original image being loaded.

Dynamic loading with Linux

Instead of Linux automatically loading and linking libraries for a given program, it's possible to share this control with the application itself. In this case, the process is called *dynamic loading.* With dynamic loading, the application can specify a particular library to load, and then use this library as an executable (that is, call the functions within it). But as you learned earlier, the shared library used for dynamic loading is no different than that of a standard shared library (an ELF shared object). In fact, the `ld-linux` dynamic linker remains involved in this process as the ELF loader and interpreter.

The Dynamic Loading (DL) API exists for dynamic loading and allows a shared library to be available to a user-space program. Although small, the API provides everything needed, with much of the hard work done behind the scenes. The full API is shown in Table 1.

Table 1. The Dl API

Function	Description
dlopen	Makes an object file accessible to a program
dlsym	Obtains the address of a symbol within a `dlopen`ed object file
dlerror	Returns a string error of the last error that occurred
dlclose	Closes an object file

The process begins with a call to `dlopen`, providing the file object to access and a mode. The result of the `dlopen` call is a handle to the object that will be used later. The `mode` argument tells the dynamic linker when to perform relocations. There are two possible values. The first, `RTLD_NOW`, indicates that the dynamic linker will complete all necessary relocations at the `dlopen` call time. The second and alternative mode, `RTLD_LAZY`, says to perform relocations only when they're needed. This is done internally by redirecting all requests that are yet to be relocated through the dynamic linker. In this way, the dynamic linker knows at request time when a new reference is occurring, and relocation occurs normally. Subsequent calls do not require a repeat of the relocation.

Two other mode options are available that may be bitwise `OR`ed into the `mode` argument. `RTLD_LOCAL` indicates that the symbols of the shared object being loaded won't be made available for relocation processing by any other object. If this is what you want (for example, so that the shared object can invoke symbols in the original process image), use `RTLD_GLOBAL`.

The `dlopen` function also automatically resolves dependencies in shared libraries. In this way, if you open an object that is dependent upon other shared libraries, it automatically loads them. The function returns a handle that is used in subsequent calls to the API. The prototype for `dlopen` is:

```
#include <dlfcn.h>

void *dlopen( const char *file, int mode );
```

With a handle to the ELF object, you can identify addresses to symbols within this object using the `dlsym` call. This function takes a symbol name, such as the name of a function contained within the object. The return value is a resolved address to the symbol within the object:

```
void *dlsym( void *restrict handle, const char *restrict name );
```

If an error occurs during a call with this API, you can use the `dlerror` function to return a human-readable string representing the error. This function has no arguments and returns a string if a prior error occurred or returns NULL if no error occurred:

```
char *dlerror();
```

Finally, when no additional calls to the shared object are necessary, the application can call `dlclose` to inform the operating system that the handle and object references are no longer necessary. This is properly reference-counted, so that multiple users of a shared object do not conflict with one another (it remains in memory as long as there is a user for it). Any symbols resolved through `dlsym` for the closed object will no longer be available.

```
char *dlclose( void *handle );
```

Dynamic loading example

Now that you've seen the API, let's look at an example of the DL API. In this application, you basically implement a shell that allows the operator to specify a library, a function, and an argument. In other words, the user can specify a library and call an arbitrary function within that library (that wasn't previously linked to this application). You resolve the function within the library using the DL API, and then call it with the user-defined argument (emitting the result). The complete application is shown in Listing 2.

Listing 2. Shell for using the DL API

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

#define MAX_STRING      80


void invoke_method( char *lib, char *method, float argument )
{
  void *dl_handle;
  float (*func)(float);
  char *error;

  /* Open the shared object */
  dl_handle = dlopen( lib, RTLD_LAZY );
  if (!dl_handle) {
    printf( "!!! %s\n", dlerror() );
    return;
  }

  /* Resolve the symbol (method) from the object */
  func = dlsym( dl_handle, method );
  error = dlerror();
  if (error != NULL) {
    printf( "!!! %s\n", error );
    return;
  }

  /* Call the resolved method and print the result */
  printf("  %f\n", (*func)(argument) );

  /* Close the object */
  dlclose( dl_handle );
  return;
}


int main( int argc, char *argv[] )
{
  char line[MAX_STRING+1];
  char lib[MAX_STRING+1];
  char method[MAX_STRING+1];
  float argument;

  while (1) {

    printf("> ");

    line[0]=0;
    fgets( line, MAX_STRING, stdin);

    if (!strncmp(line, "bye", 3)) break;

    sscanf( line, "%s %s %f", lib, method, &argument);

    invoke_method( lib, method, argument );

  }

}
```

To build this application, use the following compile line with the GNU Compiler Collection (GCC). The option `-rdynamic` is used to tell the linker to add all symbols to the dynamic symbol table (to permit backtraces with the use of `dlopen`). The `-ldl` indicates that the `dllib` should be linked to this program.

```
gcc -rdynamic -o dl dl.c -ldl
```

Back to Listing 2, the `main` function simply acts as the interpreter, parsing three arguments from the input line (library name, function name, floating-point argument). If `bye` is present, the application exits. Otherwise, the three arguments are passed to the `invoke_method` function, which uses the DL API.

You start with a call to `dlopen` to gain access to the object file. If a NULL handle is returned, the object could not be found and the process ends. Otherwise, you have a handle to the object that can be further interrogated. Using the `dlsym` API function, attempt to resolve the symbol within the newly opened object file. You'll get either a valid pointer to the symbol or a NULL and return an error.

With the symbol resolved in the ELF object, the next step is simply to call the function. Note the difference between this code and the previous discussion of dynamic linking. In this example, you coerce the address of the symbol in the object file to a function pointer, and then call it. The previous example used the object's name as a function, and the dynamic linker ensures that the symbol points to the proper location. Although the dynamic linker can do all the dirty work for you, this approach allows you to build very dynamic applications that can be extended at run time.

After you've called your target function in the ELF object, close access to it through a call to `dlclose`.

An example of how to use this test program is shown in Listing 3. In this example, you compile and then execute the program. Then, you invoke a few functions within the math library (libm.so). From this demonstration, the program is able to call arbitrary functions within a shared object (library) using dynamic loading. This is a powerful capability and permits the extension of programs with new functionality.

Listing 3. Using the simple program to invoke library functions

```
mtj@camus:~/dl$ gcc -rdynamic -o dl dl.c -ldl
mtj@camus:~/dl$ ./dl
> libm.so cosf 0.0
  1.000000
> libm.so sinf 0.0
  0.000000
> libm.so tanf 1.0
  1.557408
> bye
mtj@camus:~/dl$
```

Tools

Linux provides a variety of tools for viewing and parsing ELF objects (including shared libraries). One of the most useful is the `ldd` command, which you use to emit shared library dependencies. For example, using the `ldd` command on your `dl` application shows the following:

```
mtj@camus:~/dl$ ldd dl
        linux-gate.so.1 =>  (0xffffe000)
        libdl.so.2 => /lib/tls/i686/cmov/libdl.so.2 (0xb7fdb000)
        libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7eac000)
        /lib/ld-linux.so.2 (0xb7fe7000)
mtj@camus:~/dl$
```

What `ldd` is telling you is that this ELF image is dependent upon linux-gate.so (a special shared object that handles system calls and has no associated file in the file system), libdl.so (the DL API), the GNU `C` library (libc.so), and finally the Linux dynamic loader (as there are shared library dependencies).

The `readelf` command is a feature-rich utility that allows you to parse and read ELF objects. One interesting use of `readelf` is to identify the relocatable items within an object. For our simple program (shown in Listing 2), you can see the symbols that require relocation as:

```
mtj@camus:~/dl$ readelf -r dl

Relocation section '.rel.dyn' at offset 0x520 contains 2 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
08049a3c  00001806 R_386_GLOB_DAT    00000000   __gmon_start__
08049a78  00001405 R_386_COPY        08049a78   stdin

Relocation section '.rel.plt' at offset 0x530 contains 8 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
08049a4c  00000207 R_386_JUMP_SLOT   00000000   dlsym
08049a50  00000607 R_386_JUMP_SLOT   00000000   fgets
08049a54  00000b07 R_386_JUMP_SLOT   00000000   dlerror
08049a58  00000c07 R_386_JUMP_SLOT   00000000   __libc_start_main
08049a5c  00000e07 R_386_JUMP_SLOT   00000000   printf
08049a60  00001007 R_386_JUMP_SLOT   00000000   dlclose
08049a64  00001107 R_386_JUMP_SLOT   00000000   sscanf
08049a68  00001907 R_386_JUMP_SLOT   00000000   dlopen
mtj@camus:~/dl$
```

From this list, you can see the various `C` library calls that require relocation (to libc.so), including calls to the DL API (libdl.so). The function `__libc_start_main` is a `C` library function that is called prior to the `main` function of your program (a shell that provides necessary initialization).

Other utilities that operate on object files include `objdump`, which displays information about object files, and `nm`, which lists the symbols from object files (including debug information). It's also possible to invoke the Linux dynamic linker directly with the ELF program as its argument to manually start the image:

```
mtj@camus:~/dl$ /lib/ld-linux.so.2 ./dl
> libm.so expf 0.0
  1.000000
>
```

Additionally, you can use ld-linux.so to list the dependencies of an ELF image (identically to the `ldd` command) by using the `--list` option. Remember, it's just a user-space program that's bootstrapped by the kernel when needed.

Going further

This article scratched the surface of some of the capabilities of the dynamic linker. In the Related topics below, you'll find more detailed introductions to the ELF image format and the process or symbol relocation. And, as is always the case with Linux, you can download the source to the dynamic linker (see Related topics) to dig into its internals.

Related topics

- Download the source for the Linux dynamic linker from Debian. This is the ultimate source of information for dynamic linking and dynamic loading.
- SkyFree.org provides a great introduction to ELF (PDF) covering object files, program loading, and the `C` library. Wikipedia also provides a short description of ELF and many links to additional resources for ELF (specifications and interfaces for many processor architectures).
- The Chris Rohlf's EM_386 blog gives a detailed description of ELF symbol resolution and all its gory details. It explains the GOT and PLT tables and their manipulation by the Linux dynamic linker.
- Wikipedia has good resources on libraries and static libraries. You can also learn about linkers and loaders and their relationship to libraries.
- The *Linux Journal* article "Linkers and Loaders" (November 2002) provides a great introduction to the purpose behind linkers and loaders using ELF files (including symbol resolution and relocation).
- See all Linux tips and Linux tutorials on developerWorks.