WIKIPEDIA

Dynamic loading

Dynamic loading is a mechanism by which a computer program can, at run time, load a library (or other binary) into memory, retrieve the addresses of functions and variables contained in the library, execute those functions or access those variables, and unload the library from memory. It is one of the 3 mechanisms by which a computer program can use some other software; the other two are static linking and dynamic linking. Unlike static linking and dynamic linking, dynamic loading allows a computer program to start up in the absence of these libraries, to discover available libraries, and to potentially gain additional functionality.[1][2]

Contents

1 **History**

2 **Uses**

3 **In C/C++**

- 3.1 Summary
- 3.2 Loading the library
  - 3.2.1 Most UNIX-like operating systems (Solaris, Linux, *BSD, etc.)
  - 3.2.2 macOS
  - 3.2.3 Windows
- 3.3 Extracting library contents
  - 3.3.1 UNIX-like operating systems (Solaris, Linux, *BSD, macOS, etc.)
  - 3.3.2 Windows
- 3.4 Converting extracted library contents
  - 3.4.1 Windows
  - 3.4.2 UNIX (POSIX)
  - 3.4.3 Solving the function pointer problem on POSIX systems
- 3.5 Unloading the library
  - 3.5.1 UNIX-like operating systems (Solaris, Linux, *BSD, macOS, etc.)
  - 3.5.2 Windows
- 3.6 Special library
  - 3.6.1 UNIX-like operating systems (Solaris, Linux, *BSD, macOS, etc.)
  - 3.6.2 Windows

4 **In Java**

5 **Platforms without dynamic loading**

6 **See also**

7 External links
8 References

History

Dynamic loading was a common technique for IBM's operating systems for System/360 such as OS/360, particularly for I/O subroutines, and for COBOL and PL/I runtime libraries, and continues to be used in IBM's operating systems for z/Architecture, such as z/OS. As far as the application programmer is concerned, the loading is largely transparent, since it is mostly handled by the operating system (or its I/O subsystem). The main advantages are:

- Fixes (patches) to the subsystems fixed all programs at once, without the need to relink them
- Libraries could be protected from unauthorized modification

IBM's strategic transaction processing system, CICS (1970s onwards) uses dynamic loading extensively both for its kernel and for normal application program loading. Corrections to application programs could be made offline and new copies of changed programs loaded dynamically without needing to restart CICS[3][4] (which can, and frequently does, run 24/7).

Shared libraries were added to Unix in the 1980s, but initially without the ability to let a program load additional libraries after startup.[5]

Uses

Dynamic loading is most frequently used in implementing software plugins.[1] For example, the Apache Web Server's `*.dso` "dynamic shared object" plugin files are libraries which are loaded at runtime with dynamic loading.[6] Dynamic loading is also used in implementing computer programs where multiple different libraries may supply the requisite functionality and where the user has the option to select which library or libraries to provide.

In C/C++

Not all systems support dynamic loading. UNIX-like operating systems such as macOS, Linux, and Solaris provide dynamic loading with the C programming language "dl" library. The Windows operating system provides dynamic loading through the Windows API.

Summary

Name	Standard POSIX/UNIX API	Microsoft Windows API
Header file inclusion	`#include <dlfcn.h>`	`#include <windows.h>`
Definitions for header	`dl` (`libdl.so`, `libdl.dylib`, etc. depending on the OS)	`kernel32.dll`
Loading the library	`dlopen`	`LoadLibrary` `LoadLibraryEx`
Extracting contents	`dlsym`	`GetProcAddress`
Unloading the library	`dlclose`	`FreeLibrary`

Loading the library

Loading the library is accomplished with `LoadLibrary` or `LoadLibraryEx` on Windows and with `dlopen` on UNIX-like operating systems. Examples follow:

Most UNIX-like operating systems (Solaris, Linux, *BSD, etc.)

```
void* sdl_library = dlopen("libSDL.so", RTLD_LAZY);
if (sdl_library == NULL) {
   // report error ...
} else {
   // use the result in a call to dlsym
}
```

macOS

As a UNIX library:

```
void* sdl_library = dlopen("libsdl.dylib", RTLD_LAZY);
if (sdl_library == NULL) {
   // report error ...
} else {
   // use the result in a call to dlsym
}
```

As an macOS Framework:

```
void* sdl_library = dlopen("/Library/Frameworks/SDL.framework/SDL", RTLD_LAZY);
if (sdl_library == NULL) {
   // report error ...
} else {
   // use the result in a call to dlsym
}
```

Or if the framework or bundle contains Objective-C code:

```
NSBundle *bundle = [NSBundle bundleWithPath:@"/Library/Plugins/Plugin.bundle"];
NSError *err = nil;
if ([bundle loadAndReturnError:&err])
{
    // Use the classes and functions in the bundle.
}
else
{
    // Handle error.
}
```

Windows

```
HMODULE sdl_library = LoadLibrary("SDL.dll");
if (sdl_library == NULL) {
   // report error ...
} else {
   // use the result in a call to GetProcAddress
}
```

Extracting library contents

Extracting the contents of a dynamically loaded library is achieved with `GetProcAddress` on Windows and with `dlsym` on UNIX-like operating systems.

UNIX-like operating systems (Solaris, Linux, *BSD, macOS, etc.)

```
void* initializer = dlsym(sdl_library,"SDL_Init");
if (initializer == NULL) {
   // report error ...
} else {
   // cast initializer to its proper type and use
}
```

On macOS, when using Objective-C bundles, one can also:

```
Class rootClass = [bundle principalClass]; // Alternatively, NSClassFromString() can be used to obtain
if (rootClass)
{
    id object = [[rootClass alloc] init]; // Use the object.
}
else
{
    // Report error.
}
```

Windows

```
FARPROC initializer = GetProcAddress(sdl_library,"SDL_Init");
if (initializer == NULL) {
   // report error ...
} else {
   // cast initializer to its proper type and use
}
```

Converting extracted library contents

The result of `dlsym()` or `GetProcAddress()` has to be converted to the desired destination before it can be used.

Windows

In the Windows case, the conversion is straightforward, since FARPROC is essentially already a function pointer:

```
typedef INT_PTR (*FARPROC)(void);
```

This can be problematic when the address of an object is to be retrieved rather than a function. However, usually one wants to extract functions anyway, so this is normally not a problem.

```
typedef void (*sdl_init_function_type)(void);
sdl_init_function_type init_func = (sdl_init_function_type) initializer;
```

UNIX (POSIX)

According to the POSIX specification, the result of `dlsym()` is a `void` pointer. However, a function pointer is not required to even have the same size as a data object pointer, and therefore a valid conversion between type `void*` and a pointer to a function may not be easy to implement on all platforms.

On most systems in use today, function and object pointers are *de facto* convertible. The following code snippet demonstrates one workaround which allows to perform the conversion anyway on many systems:

```
typedef void (*sdl_init_function_type)(void);
sdl_init_function_type init_func = (sdl_init_function_type)initializer;
```

The above snippet will give a warning on some compilers: `warning: dereferencing type-punned pointer will break strict-aliasing rules`. Another workaround is:

```
typedef void (*sdl_init_function_type)(void);
union { sdl_init_function_type func; void * obj; } alias;
```

```
alias.obj = initializer;
sdl_init_function_type init_func = alias.func;
```

which disables the warning even if strict aliasing is in effect. This makes use of the fact that reading from a different union member than the one most recently written to (called "type punning") is common, and explicitly allowed even if strict aliasing is in force, provided the memory is accessed through the union type directly.[7] However, this is not strictly the case here, since the function pointer is copied to be used outside the union. Note that this trick may not work on platforms where the size of data pointers and the size of function pointers is not the same.

Solving the function pointer problem on POSIX systems

The fact remains that any conversion between function and data object pointers has to be regarded as an (inherently non-portable) implementation extension, and that no "correct" way for a direct conversion exists, since in this regard the POSIX and ISO standards contradict each other.

Because of this problem, the POSIX documentation on `dlsym()` for the outdated issue 6 stated that "a future version may either add a new function to return function pointers, or the current interface may be deprecated in favor of two new functions: one that returns data pointers and the other that returns function pointers".[8]

For the subsequent version of the standard (issue 7, 2008), the problem has been discussed and the conclusion was that function pointers have to be convertible to `void*` for POSIX compliance.[9] This requires compiler makers to implement a working cast for this case.

If the contents of the library can be changed (i.e. in the case of a custom library), in addition to the function itself a pointer to it can be exported. Since a pointer to a function pointer is itself an object pointer, this pointer can always be legally retrieved by call to `dlsym()` and subsequent conversion. However, this approach requires maintaining separate pointers to all functions that are to be used externally, and the benefits are usually small.

Unloading the library

Loading a library causes memory to be allocated; the library must be deallocated in order to avoid a memory leak. Additionally, failure to unload a library can prevent filesystem operations on the file which contains the library. Unloading the library is accomplished with `FreeLibrary` on Windows and with `dlclose` on UNIX-like operating systems. However, unloading a DLL can lead to program crashes if objects in the main application refer to memory allocated within the DLL. For example, if a DLL introduces a new class and the DLL is closed, further operations on instances of that class from the main application will likely cause a memory access violation. Likewise, if the DLL introduces a factory function for instantiating dynamically loaded classes, calling or dereferencing that function after the DLL is closed leads to undefined behaviour.

UNIX-like operating systems (Solaris, Linux, *BSD, macOS, etc.)

```
dlclose(sdl_library);
```

Windows

```
FreeLibrary(sdl_library);
```

Special library

The implementations of dynamic loading on UNIX-like operating systems and Windows allow programmers to extract symbols from the currently executing process.

UNIX-like operating systems allow programmers to access the global symbol table, which includes both the main executable and subsequently loaded dynamic libraries.

Windows allows programmers to access symbols exported by the main executable. Windows does not use a global symbol table, and has no API to search across multiple modules to find a symbol by name.

UNIX-like operating systems (Solaris, Linux, *BSD, macOS, etc.)

```
void* this_process = dlopen(NULL,0);
```

Windows

```
HMODULE this_process = GetModuleHandle(NULL);

HMODULE this_process_again;
GetModuleHandleEx(0,0,&this_process_again);
```

In Java

In the Java programming language, classes can be dynamically loaded using the **`ClassLoader (https://docs.oracle.com/javase/9/docs/api/java/lang/ClassLoader.html)`** object. For example:

```
Class type = ClassLoader.getSystemClassLoader().loadClass(name);
Object obj = type.newInstance();
```

The Reflection mechanism also provides a means to load a class if it isn't already loaded. It uses the classloader of the current class:

```
Class type = Class.forName(name);
Object obj = type.newInstance();
```

However, there is no simple way to unload a class in a controlled way. Loaded classes can only be unloaded in a controlled way, i.e. when the programmer wants this to happen, if the classloader used to load the class is not the system class loader, and is itself unloaded. When doing so, various details need to be observed to ensure the class is really unloaded. This makes unloading of classes tedious.

Implicit unloading of classes, i.e. in an uncontrolled way by the garbage collector, has changed a few times in Java. Until Java 1.2. the garbage collector could unload a class whenever it felt it needed the space, independent of which class loader was used to load the class. Starting with Java 1.2 classes loaded via the system classloader were never unloaded and classes loaded via other classloaders only when this other classloader was unloaded. Starting with Java 6 classes can contain an internal marker indicating to the garbage collector they can be unloaded if the garbage collector desires to do so, independent of the classloader used to load the class. The garbage collector is free to ignore this hint.

Similarly, libraries implementing native methods are dynamically loaded using the `System.loadLibrary` method. There is no `System.unloadLibrary` method.

Platforms without dynamic loading

Despite its promulgation in the 1980s through UNIX and Windows, some systems still chose not to add—or even to remove—dynamic loading. For example, Plan 9 from Bell Labs and its successor 9front consider dynamic linking harmful, and purposefully do not support it.[10] The Go programming language, by some of the same developers as Plan 9, also did not support dynamic linking, but plugin loading is available since Go 1.8 (https://tip.golang.org/doc/go1.8) (February 2017). The Go runtime and any library functions are statically linked into the compiled binary.[11]

See also

- Compile and go system
- DLL Hell
- Direct binding
- Dynamic binding (computing)
- Dynamic dispatch
- Dynamic library
- Dynamic linker
- Dynamic-link library
- GNU linker
- Library (computing)
- Linker (computing)
- Loader (computing)
- Name decoration
- Prebinding
- Prelinking
- Relocation (computer science)
- Relocation table
- Static library
- gold (linker)

- prelink

External links

- General Links
  - Dynamic Loading (http://linux4u.jinr.ru/usoft/WWW/www_debian.org/Documentation/elf/node7.html) on Linux4U
  - Dynamic Shared Object (DSO) Support (http://httpd.apache.org/docs/current/dso.html) by Apache
  - C++ Dynamic Linking By Example (http://www.informit.com/guides/content.aspx?g=cplusplus&seqNum=153)
  - Dynamic Library Loading Example (https://github.com/danfuzz/dl-example) (complete but concise working example)
  - Dynamic Library Programming Topics from Apple Developer Connection (targeted to macOS) (https://developer.apple.com/library/mac/#DOCUMENTATION/DeveloperTools/Conceptual/DynamicLibraries/000-Introduction/Introduction.html#//apple_ref/doc/uid/TP40001908-SW1)
- C/C++ UNIX API:
  - dlopen (http://www.opengroup.org/onlinepubs/009695399/functions/dlopen.html)
  - dlsym (http://www.opengroup.org/onlinepubs/009695399/functions/dlsym.html)
  - dlclose (http://www.opengroup.org/onlinepubs/009695399/functions/dlclose.html)
- C/C++ Windows API:
  - LoadLibrary (http://msdn2.microsoft.com/en-us/library/ms684175.aspx)
  - GetProcAddress (http://msdn2.microsoft.com/en-us/library/ms683212(VS.85).aspx)
  - FreeLibrary (http://msdn2.microsoft.com/en-us/library/ms683152(VS.85).aspx)
  - Delay-Loaded DLLs (http://msdn.microsoft.com/en-us/library/151kt790.aspx)
- Java API:
  - ClassLoader (http://java.sun.com/j2se/1.5.0/docs/api/java/lang/ClassLoader.html)
  - Class (http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Class.html)

References

1. Autoconf, Automake, and Libtool: Dynamic Loading (http://sourceware.org/autobook/autobook/autobook_158.html)
2. Linux4U: ELF Dynamic Loading (http://linux4u.jinr.ru/usoft/WWW/www_debian.org/Documentation/elf/node7.html)
3. http://publib.boulder.ibm.com/infocenter/cicsts/v3r1/index.jsp?topic=/com.ibm.cics.ts31.doc/dfhp3/dfhp3oq.htm
4. http://www-01.ibm.com/support/docview.wss?uid=swg21031546

5. W. Wilson Ho; Ronald A. Olsson (1991). "An approach to genuine dynamic linking". *Software —Practice and Experience*. **21** (4): 375–390. doi:10.1002/spe.4380210404 (https://doi.org/10.1002%2Fspe.4380210404).
6. Apache 1.3 Dynamic Shared Object (DSO) Support (http://httpd.apache.org/docs/1.3/dso.html)
7. GCC 4.3.2 Optimize Options: -fstrict-aliasing (https://gcc.gnu.org/onlinedocs/gcc-4.3.2/gcc/Optimize-Options.html#index-fstrict_002daliasing-721)
8. POSIX documentation on `dlopen()` (http://www.opengroup.org/onlinepubs/009695399/functions/dlsym.html) (issue 6).
9. POSIX documentation on `dlopen()` (http://www.opengroup.org/onlinepubs/9699919799/functions/dlsym.html) (issue 7)
10. "Dynamic Linking" (http://harmful.cat-v.org/software/dynamic-linking/). *cat-v.org*. 9front. Retrieved 22 December 2014.
11. "Go FAQ" (http://golang.org/doc/faq#Why_is_my_trivial_program_such_a_large_binary).

- The two subsections 8.1.4 "Dynamic Loading" and 8.1.5 "Dynamic Linking and shared libraries" in Silberschatz, Abraham; Galvin, Peter Baer; Gagne, Greg (2005). *Operating System Concepts*. J. Wiley & Sons. ISBN 0-471-69466-5.