Overview of Dynamic Libraries

Published: 2012-07-23 · Archived: 2026-04-06 00:08:25 UTC

Two important factors that determine the performance of apps are their launch times and their memory footprints. Reducing the size of an app's executable file and minimizing its use of memory once it's launched make the app launch faster and use less memory once it's launched. Using dynamic libraries instead of static libraries reduces the executable file size of an app. They also allow apps to delay loading libraries with special functionality only when they're needed instead of at launch time. This feature contributes further to reduced launch times and efficient memory use.

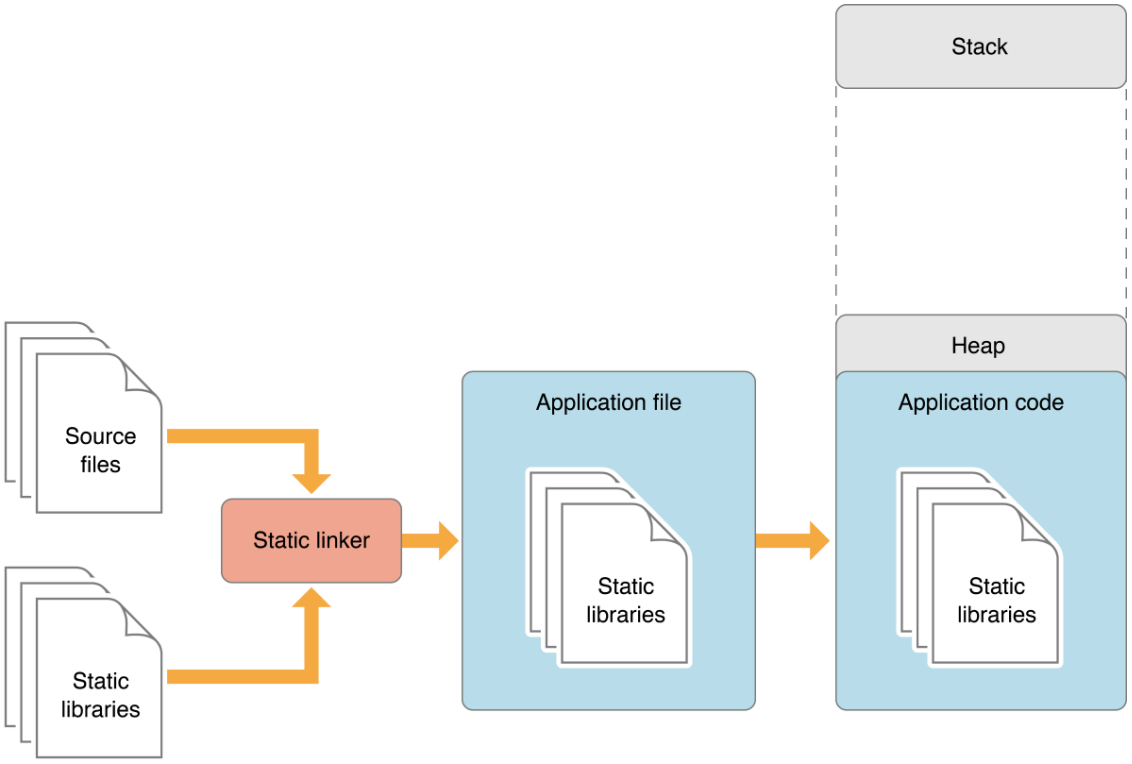
This article introduces dynamic libraries and shows how using dynamic libraries instead of static libraries reduces both the file size and initial memory footprint of the apps that use them. This article also provides an overview of the dynamic loader compatibility functions apps use to work with dynamic libraries at runtime.

What Are Dynamic Libraries?

Most of an app's functionality is implemented in libraries of executable code. When an app is linked with a library using a static linker, the code that the app uses is copied to the generated executable file. A *static linker* collects compiled source code, known as object code, and library code into one executable file that is loaded into memory in its entirety at runtime. The kind of library that becomes part of an app's executable file is known as a static library. *Static libraries* are collections or archives of object files.

When an app is launched, the app's code—which includes the code of the static libraries it was linked with—is loaded into the app's address space. Linking many static libraries into an app produces large app executable files. Figure 1 shows the memory usage of an app that uses functionality implemented in static libraries. Applications with large executables suffer from slow launch times and large memory footprints. Also, when a static library is updated, its client apps don't benefit from the improvements made to it. To gain access to the improved functionality, the app's developer must link the app's object files with the new version of the library. And the apps users would have to replace their copy of the app with the latest version. Therefore, keeping an app up to date with the latest functionality provided by static libraries requires disruptive work by both developers and end users.

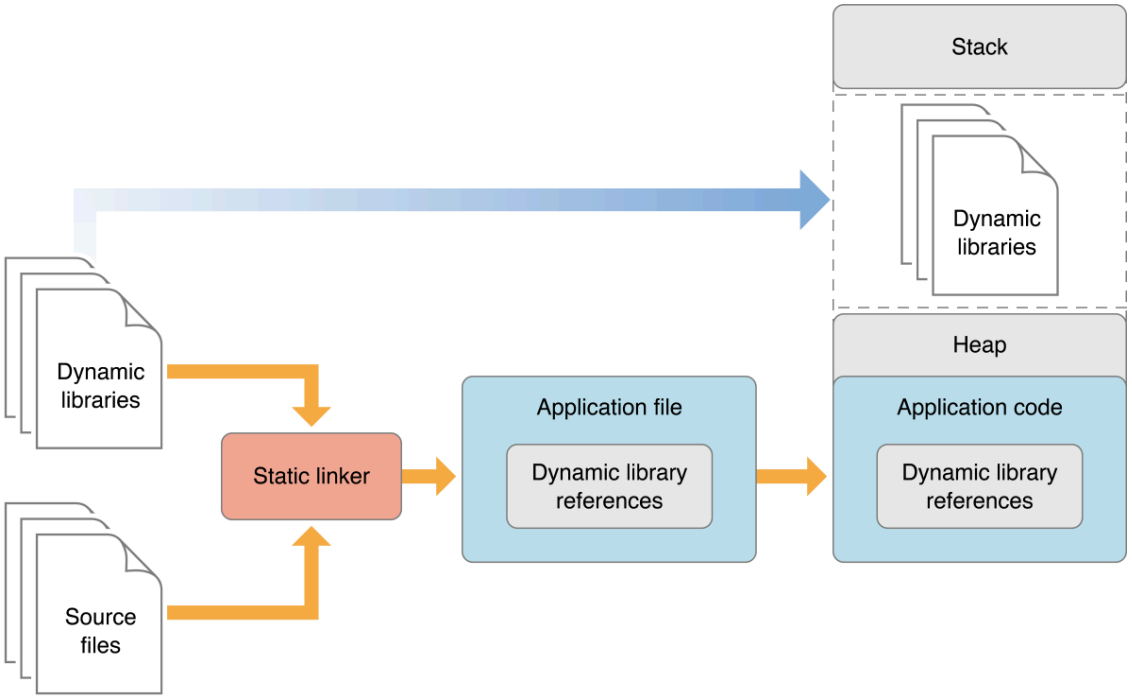
Figure 1 App using static libraries



A better approach is for an app to load code into its address space when it's actually needed, either at launch time or at runtime. The type of library that provides this flexibility is called *dynamic library*. Dynamic libraries are not statically linked into client apps; they don't become part of the executable file. Instead, dynamic libraries can be loaded (and linked) into an app either when the app is launched or as it runs.

Figure 2 shows how implementing some functionality as dynamic libraries instead of as static libraries reduces the memory used by the app after launch.

Figure 2 App using dynamic libraries



Using dynamic libraries, programs can benefit from improvements to the libraries they use automatically because their link to the libraries is dynamic, not static. That is, the functionality of the client apps can be improved and extended without requiring app developers to recompile the apps. Apps written for OS X benefit from this feature because all system libraries in OS X are dynamic libraries. This is how apps that use Carbon or Cocoa technologies benefit from improvements to OS X.

Another benefit dynamic libraries offer is that they can be initialized when they are loaded and can perform clean-up tasks when the client app terminates normally. Static libraries don't have this feature. For details, see Module Initializers and Finalizers.

One issue that developers must keep in mind when developing dynamic libraries is maintaining compatibility with client apps as a library is updated. Because a library can be updated without the knowledge of the client-app's developer, the app must be able to use the new version of the library without changes to its code. To that end, the library's API should not change. However, there are times when improvements require API changes. In that case, the previous version of the library must remain in the user's computer for the client app to run properly. Dynamic Library Design Guidelines explores the subject of managing compatibility with client apps as a dynamic library evolves.

How Dynamic Libraries Are Used

When an app is launched, the OS X kernel loads the app's code and data into the address space of a new process. The kernel also loads the dynamic loader (`/usr/lib/dyld`) into the process and passes control to it. The dynamic loader then loads the app's *dependent libraries*. These are the dynamic libraries the app was linked with. The static linker records the filenames of each of the dependent libraries at the time the app is linked. This filename is known as the dynamic library's *install name*. The dynamic loader uses the app's dependent libraries' install names to locate them in the file system. If the dynamic loader doesn't find all the app's dependent libraries at launch time or if any of the libraries is not compatible with the app, the launch process is aborted. For more information on dependent-library compatibility, see Managing Client Compatibility With Dependent Libraries. Dynamic library developers can set a different install name for a library when they compile it using the `gcc -install_name` option. See the `gcc` man page for details.

The dynamic loader resolves only the undefined external symbols the app actually uses during the launch process. Other symbols remain unresolved until the app uses them. For details on the process the dynamic loader goes when an app is launched, see "Executing Mach-O Files" in *Mach-O Programming Topics*.

The dynamic loader—in addition to automatically loading dynamic libraries at launch time—loads dynamic libraries at runtime, at the app's request. That is, if an app doesn't require that a dynamic library be loaded when it launches, developers can choose to not link the app's object files with the dynamic library, and, instead, load the dynamic library only in the parts of the app that require it. Using dynamic libraries this way speeds up the launch process. Dynamic libraries loaded at runtime are known as *dynamically loaded libraries*. To load libraries at runtime, apps can use functions that interact with the dynamic loader for the platform under which they're running.

Different platforms implement their dynamic loaders differently. They may also have custom dynamic code-loading interfaces that make code difficult to port across platforms. To facilitate porting an app from UNIX to

Linux, for example, Jorge Acereda and Peter O'Gorman developed the *dynamic loader compatibility (DLC)* functions. They offer developers a standard, portable way to use dynamic libraries in their apps.

The DLC functions are declared in `/usr/include/dlfcn.h`. There are five of them:

- `dlopen(3) OS X Developer Tools Manual Page`: Opens a dynamic library. An app calls this function before using any of the library's exported symbols. If the dynamic library hasn't been opened by the current process, the library is loaded into the process's address space. The function returns a handle that's used to refer to the opened library in calls to `dlsym` and `dlclose`. This handle is known as the *dynamic library handle*. This function maintains a reference count that indicates the number of times the current process has used `dlopen` to open a particular dynamic library.

- `dlsym(3) OS X Developer Tools Manual Page`: Returns the address of a symbol exported by a dynamically loaded library. An app calls this function after obtaining a handle to the library through a call to `dlopen`. The `dlsym` function takes as parameters the handle returned by `dlopen` or a constant specifying the symbol search scope and the symbol name.

- `dladdr(3) OS X Developer Tools Manual Page`: Returns information on the address provided. If the address corresponds to a dynamically loaded library within the app's address space, this function returns information on the address. This information is returned in a `Dl_info` structure, which encapsulates the pathname of the dynamic library, the library's base address, and the address and value of the nearest symbol to the address provided. If no dynamic library is found at the address provided, the function returns no information.

- `dlclose(3) OS X Developer Tools Manual Page`: Closes a dynamically loaded library. This function takes as a parameter a handle returned by `dlopen`. When the reference count for that handle reaches 0, the library is unloaded from the current process's address space.

- `dlerror(3) OS X Developer Tools Manual Page`: Returns a string that describes an error condition encountered by the last call to `dlopen`, `dlsym`, or `dlclose`.

For more information on the DLC functions, see *OS X ABI Dynamic Loader Reference*.
