Executing Mach-O Files

To perform their objectives, programs must execute processes and link to dynamic shared libraries. To work with other libraries or modules, your application must define references to symbols in those modules; those references are resolved at runtime. At runtime the symbol names of all the modules your application uses live in a shared namespace, similar to a directory. To allow for future enhancements to applications as well as the libraries they use, application and library developers must ensure the names they choose for their functions and data do not conflict with the names used in other modules.

The two-level namespace feature of OS X v10.1 and later adds the module name as part of the symbol name of the symbols defined within it. This approach ensures a module's symbol names don't conflict with the names used in other modules. To perform special tasks or to provide an enhanced user experience, your application may need to launch other applications or create processes to run command-line tools. To maintain a high degree of interoperability and provide a consistent user experience, your applications should use specific system functions and frameworks to execute processes and launch applications.

This article provides an overview of the OS X dynamic loading process. The process of loading and linking a program in OS X mainly involves two entities: the OS X kernel and the dynamic linker. When you execute a program, the kernel creates a process for the program, and loads the program and the dynamic linker shared library, usually `/usr/lib/dyld`, in the program's address space. The kernel then executes code in the dynamic linker that loads the libraries the program references. This article also describes the visibility symbols in a module get depending on how they are defined and the process of resolving symbol references at runtime.

Launching an Application

When you launch an application from the Finder or the Dock, or when you run a program in a shell, the system ultimately calls two functions on your behalf, `fork` and `execve`. The `fork` function creates a process; the `execve` function loads and executes the program. There are several variant exec functions, such as `execl`, `execv`, and `exect`, each providing a slightly different way of passing arguments and environment variables to the program. In OS X, each of these other exec routines eventually calls the kernel routine `execve`.

When writing a Mac app, you should use the Launch Services framework to launch other applications. Launch Services understands application packages, and you can use it to open both applications and documents. The Finder and the Dock use Launch Services to maintain the database of mappings from document types to the applications that can open them. Cocoa applications can use the class `NSWorkspace` to launch applications and documents; `NSWorkspace` itself uses Launch Services. Launch Services ultimately calls `fork` and `execve` to do the actual work of creating and executing the new process. For more information on Launch Services, see *Launch Services Programming Guide*.

Forking and Executing the Process

To create a process using BSD system calls, your process must call the `fork` system call. The `fork` call creates a logical copy of your process, then returns the ID of the new process to your process. Both the original process and the new process continue executing from the call to `fork`; the only difference is that `fork` returns the ID of the new process to the original process and zero to the new process. (The `fork` function returns `-1` to the original process and sets `errno` to a specific error value if the new process could not be created.)

To run a different executable, your process must call the `execve` system call with a pathname specifying the location of the alternate executable. The `execve` call replaces the program currently in memory with a different executable file.

A Mach-O executable file contains a header consisting of a set of load commands. For programs that use shared libraries or frameworks, one of these commands specifies the location of the linker to be used to load the program. If you use Xcode, this is always `/usr/lib/dyld`, the standard OS X dynamic linker.

When you call the `execve` routine, the kernel first loads the specified program file and examines the `mach_header` structure at the start of the file. The kernel verifies that the file appear to be a valid Mach-O file and interprets the load commands stored in the header. The kernel then loads the dynamic linker specified by the load commands into memory and executes the dynamic linker on the program file.

The dynamic linker loads all the shared libraries that the main program links against (the *dependent libraries*) and binds enough of the symbols to start the program. It then calls the entry point function. At build time, the static linker adds the standard entry point function to the main executable file from the object file `/usr/lib/crt1.o`. This function sets up the runtime environment state for the kernel and calls static initializers for C++ objects, initializes the Objective-C runtime, and then calls the program's `main` function.

Finding Imported Symbols

When the dynamic linker loads a Mach-O file (which, for the purposes of this section, is called the *client program*), it connects the file's imported symbols to their definitions in a shared library or framework. This section describes the process of binding the imported symbols in one Mach-O file to their definitions in other Mach-O files. It also explains the process of finding a symbol. See also Loading Plug-in Code With Bundles in Loading Code at Runtime for information on finding symbols in plug-ins.

Binding Symbols

Binding is the process of resolving a module's references to functions and data in other modules (the *undefined external symbols*, sometimes called *imported symbols*). The modules may be in the same Mach-O file or in different Mach-O files; the semantics are identical in either case. When the application is first loaded, the dynamic linker loads the imported shared libraries into the address space of the program. When binding is performed, the linker replaces each of the program's imported references with the address of the actual definition from one of the shared libraries.

The dynamic linker can bind a program at several stages during loading and execution, depending on the options you specify at build time:

- With *just-in-time binding* (also called lazy binding), the dynamic linker binds a reference (and all the other references in the same module) when the program first uses the reference. The dynamic linker loads the shared libraries the client program depends on when the program is loaded. However, the dynamic linker doesn't bind the program's references to symbols within the shared libraries until the symbols are used.
- With *load-time binding*, the dynamic linker binds all the imported references immediately upon loading the program, or, for bundles, upon loading the bundle. To use load-time binding with the standard tools, specify the `-bind_at_load` option to `ld` to indicate that the dynamic linker must immediately bind all external references when the file is loaded. Without this option, `ld` sets up the output file for just-in-time binding.
- With *prebinding*, a form of load-time binding, the shared libraries referenced by the program are each prebound at a specified address. The static linker sets the address of each undefined reference in the program to default to these addresses. At runtime, the dynamic linker needs only to verify that none of the addresses have changed since the program was built (or since the prebinding was recomputed). If the addresses have changed, the dynamic linker must undo the prebinding by clearing the prebound

addresses for all the undefined references and then proceed as if the program had been just-in-time bound. Otherwise, it does not need to perform any action to bind the program.

Prebinding requires that each framework specify its desired base virtual memory address and that none of the prebound addresses of the loaded frameworks overlap. To prebind a file with the standard tools, specify the `-prebind` option to `ld`.

- *Weak references*, a feature introduced in OS X v10.2, is useful for selectively implementing features that may be available on some systems, but not on others. This mode of binding allows a program to optionally bind to specified shared libraries. If the dynamic linker cannot find definitions for weak references, it sets them to `NULL` and continues to load the program. The program can check at runtime to find out whether or not a reference is null and, if so, avoid using the reference. You can specify both libraries and individual symbols to be weakly referenced.

Note: The OS X weak linking design is derived from the classic Mac OS Code Fragment Manager implementation of weak linking. If you are familiar with the ELF executable format, you may be used to a different meaning for the terms *weak symbol* or *weak linking*, where a weak symbol may be overridden by a non-weak symbol. The equivalent OS X feature is the *weak definition*—see Scope and Treatment of Symbol Definitions for more information

If no other type of binding is specified for a given library, the static linker sets up the program's undefined references to that library to use just-in-time binding.

Searching for Symbols

A *symbol* is a generic representation of the location of a function, data variable, or constant in an executable file. References to functions and data in a program are references to symbols. To refer to a symbol when using the dynamic linking routines, you usually pass the name of the symbol, although some functions also accept a number representing the ordering of the symbol in the executable file. The name of a symbol representing a function that conforms to standard C calling conventions is the name of the function with an underscore prefix. Thus, the name of the symbol representing the function `main` would be `_main`.

Programs created by the OS X v10.0 development tools add all symbols from all loaded shared libraries into a single global list. Any symbol that your program references can be located in any shared library, as long as that shared library is one of the program's dependent libraries (or one of the dependent libraries of the dependent libraries).

OS X v10.1 introduced the two-level symbol namespace feature. The first level of the two-level namespace is the name of the library that contains the symbol, and the second is the name of the symbol. With the two-level namespace feature enabled, when the static linker records references to imported symbols, it records a reference to the name of the library that contains the symbol and the name of the symbol. Linking your programs with the two level namespace feature offers two benefits over the flat namespace:

- *Enhanced performance when searching for symbols.* With the two-level namespace, the dynamic linker knows exactly where to start looking for the implementation of a symbol. With a flat namespace, the dynamic linker must search all the loaded libraries for the one that contains the symbol.
- *Enhanced forward compatibility.* In the flat namespace, two or more libraries cannot contain symbols with different implementations that share the same name because the dynamic linker cannot know which library contains the preferred implementation. This is not initially a problem, because the static linker catches any such problems when you first build the application. However, if the vendor of one of your dependent shared libraries later releases a new version of the library that contains a symbol with the same name as one in your program or in another dependent shared library, your program will fail to run.

Your application must link directly to the shared library that contains the symbol (or, if the library is part of an umbrella framework, to the umbrella framework that contains it).

When obtaining symbols in a program built with the two-level namespace feature enabled, you must specify a reference to the shared library that contains the symbols.

By default, the static linker in OS X v10.1 and later uses a two-level namespace for all Mach-O files.

> **Note:** The OS X two-level namespace feature is loosely based on the design of the Code Fragment Manager's namespace. A two-level namespace is approximately equivalent to the namespace used to look up symbols in code fragments. Because Code Fragment Manager always requires an explicit reference to the library in which a symbol should be found, there is no Code Fragment Manager equivalent to a flat namespace search.

For programs that do not have a two-level namespace, you can tell the linker to define references to undefined symbols even if the linker cannot find the library that contains them. When you build an executable with such undefined symbols, you are making the assumption that one of the other files loaded as part of the executable file at runtime contains those symbols. Bundles and shared libraries sometimes use this option to reference symbols defined in the main executable. However, this causes you to lose the performance and compatibility benefits of two-level namespaces. It's usually better to explicitly link against an executable that defines the references. However, if you must link with undefined references, you can do it by enabling the flat namespace feature and suppressing undefined reference warnings, using the options `-flat_namespace` and `-undefined suppress` as in the following command line:

```
ld -o my_tool -flat_namespace -undefined suppress peace.o love.o
```

When building executables with a two-level namespace, you can allow the remaining undefined symbols to be looked up by the dynamic linker if the program is targeted for OS X v10.3 and later (the `MACOSX_DEPLOYMENT_TARGET` environment variable is set to `10.3` or higher). To take advantage of this feature, use the `-undefined dynamic_lookup` option.

To build executables with a two-level namespace, the static linker must be able to find the source library for each symbol. This can present difficulties for authors of bundles and dynamic shared libraries that assume a flat, global symbol namespace. To build successfully with the two-level namespace, keep the following points in mind:

- Bundles that need to reference symbols defined in the program's main executable must use the `-bundle_loader` static linker option. The static linker can then search the main executable for the undefined symbols.
- Shared libraries that need to reference symbols defined in the program's main executable must load the symbol dynamically using a function that does not require a library reference, such as `dlsym` or `NSLookupSymbolInImage` (*OS X ABI Dynamic Loader Reference*).

> **Note:** A two-level symbol namespace can be searched using functions for doing flat symbol searches.

Scope and Treatment of Symbol Definitions

Symbols in an object file may exist at several levels of scope. This section describes each of the possible scopes that a symbol may be defined at, and provides samples of C code used to create each symbol type. These samples work with the standard developer tools; a third party tool set may have different conventions.

A *defined external symbol* is any symbol defined in the current object file, including functions and data. The following C code defines external symbols:

```
int x = 0;
double y = 99 __attribute__((visibility("default")));   // GCC 4.0 only
```

An *undefined external symbol* is any symbol defined in a file outside of the current file. The following C code defines two external symbols, a variable and a function:

```
extern int x;
extern void SomeFunction(void);
```

A *common symbol* is a symbol that may appear in multiple intermediate object files. The static linker permits multiple common symbol definitions with the same name in input files, and copies the one with the largest size to the final product. If there is another symbol with the same name as a common symbol, the static linker ignores the common symbol instead.

The standard C compiler generates a common symbol when it sees a *tentative definition*—a global variable that has no initializer and is not marked `extern`. The following line is an example of a tentative definition:

```
int x;
```

A multi-module shared library, which `ld` builds by default, cannot have common symbols. However, you can build a shared library as a single module with the `-single_module` flag. To eliminate common symbols in an existing shared library, you must either explicitly define the symbol (with an initialized value, for example) in one of the modules in the shared library, or pass the `-fno-common` flag to the compiler.

A *private defined symbol* is a symbol that is not visible to other modules. The following C code defines a private symbol:

```
static int x;
```

A *private external symbol* is a defined external symbol that is visible only to other modules within the same object file as the module that contains it. The standard static linker changes private external symbols into private defined symbols unless you specify otherwise (using the `-keep_private_externs` flag).

You can mark a symbol as private external by using the `__private_extern__` keyword (which works only in C) or the `visibility("hidden")` attribute (which works both in C and C++ with GCC 4.0), as in this example:

```
__private_extern__ int x = 0;                       // C only
int y = 99 __attribute__((visibility("hidden")));   // C and C++, GCC 4.0 only
```

A *coalesced symbol* is a symbol that may be defined in multiple object files but that the static linker generates only one copy of in the output file. This can save a lot of memory with certain C++ language features that the compiler must generate for each individual object file, such as virtual function tables, runtime type information (RTTI), and C++ template instantiations. The compiler determines which constructs should be coalesced; no work on your part is required.

A *weak reference* is an undefined external symbol that need not be found in order for the client program to successfully link. If the symbol does not exist, the dynamic linker sets the address of the symbol to zero. Files with weak references can be used only in OS X v10.2 and later. The following C code demonstrates conditionalizing an API call using a weak reference:

```
/* Only call this API if it exists */
if ( SomeNewFunction != NULL )
        SomeNewFunction();
```

To specify that a function should be treated as a weak reference, use the `weak_import` attribute on a function prototype, as demonstrated by the following code:

```
void SomeNewFunction(void) __attribute__((weak_import));
```

A *coalesced weak reference* is an undefined external reference to a symbol defined in multiple object files. In OS X v10.4 and later (with GCC 4.0 and later), you can specify that a symbol be made into a coalesced weak reference by adding the weak attribute to the symbol's declaration. For example:

```
void SomeNewFunction(void) __attribute__((weak));
```

Note: Programmers who use other operating systems may be familiar with the concept of symbols that are marked with a COMDAT flag; a coalesced symbol is the OS X equivalent feature.

A *weak definition* is a symbol that is ignored by the linker if an otherwise identical but nowness definition exists. This is used by the standard C++ compiler to support C++ template instantiations. The compiler marks implicit—and not explicit—template instantiations as weak definitions. The static linker then prefers any explicit template instantiation to an implicit one for the same symbol, which provides correct C++ linking semantics. As with coalesced symbols, the compiler determines the constructs that require the weak definitions feature; no work on your part is required.

Note: Files with weak definitions can be used only in OS X v10.2 and later. The static linker changes any weak definitions into nowness definitions, so this is only a concern for intermediate object files and static libraries that you wish to deploy on earlier versions of OS X.

A *debugging symbol* is a symbol generated by the compiler that allows the debugger to map from addresses in machine code to locations in source code. The standard compilers generate debugging symbols using either the Stabs format or the DWARF format (supported in Xcode 2.4 and later). When using the Stabs format, debugging symbols, like other symbols, are stored in the symbol table (see *OS X ABI Mach-O File Format Reference*). But with the DWARF format, debugging symbols are stored in a specialized segment: the `__DWARF` segment. With DWARF you also have the option of storing debugging symbols in a separate debug-information file, which reduces the size of the binary files while permitting a full debugging experience when the corresponding debug-information files are available.