Dynamic linking best practices

July 4, 2021 **Newsletter ↳**

In this article we'll learn how to build shared libraries and install them properly on several platforms. For guidance, we'll examine the goals and history of dynamic linking on UNIX-based operating systems.

Content for the article comes from researching how to create a shared library, wading through sloppy conventions that people recommend online, and testing on multiple Unix-like systems. Hopefully it can set the record straight and help improve the quality of open source libraries.

- The common UNIX pattern
- Versioning
    - Version identifiers
    - API vs ABI
- Variance of linker and loader by system
    - Linkers (ld, lld)
    - Loaders (ld.so, dyld)
- Portable best practices
    - Linking
    - Loading
- Example code

The common UNIX pattern

The design typically used nowadays for dynamic linking (in BSD, MacOS, and Linux) came from SunOS in 1988. The paper Shared Libraries in SunOS (https://www.cs.cornell.edu/courses/cs414/2001FA/sharedlib.pdf) neatly explains the goals, design, and implementation.

The authors' main motivations were saving disk and memory space, and upgrading libraries (or the OS) without needing to relink programs. The resource usage motivation is probably less important on today's powerful personal computers than it was in 1988. However, the flexibility to upgrade libraries is as useful as ever, as well as the ability to easily inspect which library versions each application uses.

Dynamic linking is not without its critics, and isn't appropriate in all situations. It runs a little slower because of position-independent code (PIC) and late loading. (The SunOS paper called it a "classic space/time trade-off.") The complexity of the loader on some systems offers increased attack surface (http://www.nth-dimension.org.uk/pub/BTL.pdf). Finally, upgraded libraries may affect some programs differently than others, for instance breaking those that rely on undocumented behavior.

The link editor and loader

At compile time the link editor resolves symbols in specified libraries, and makes a note in the resulting binary to load those libraries. At runtime, applications call code to map the shared library symbols in memory at the correct memory addresses.

SunOS and subsequent UNIX-like systems added compile-time flags to the linker (ld) to generate – or link against – dynamically linked libraries. The designers also added a special system library (ld.so) with code to find and load other libraries for an application. The pre-`main()` initialization routine of a program loads ld.so and runs it from within the program to find and load the rest of the required libraries.

Versioning

As mentioned, applications can take advantage of updated libraries without needing recompilation. Library updates can be classified in three categories:

1. Implementation improvements for the current interface. Bug fixes, performance. (Patch release)
2. New features, additions to the interface. (Minor release)
3. Backward-incompatible change to the interface or its operation. (Major release)

An application linked against a library at a given major release will continue to work properly when loading any newer minor or patch release. Applications may not work properly when loading a different major release, or an earlier minor release than that used at link time.

Multiple applications can exist on a machine at once, and each may require different releases of a single library. The system should provide a way to store multiple library releases and load the right one for each app. Different systems have different ways to do it, as we'll see later.

Version identifiers

Each library release can be marked with a version identifier (or "version") which seeks to capture information about the library's release history. There are multiple ways to map release history to a version identifier.

The two most common mapping systems are *semantic versioning* and *libtool versioning*. Semantic versioning counts the number of releases of various kinds that have happened, and writes them in lexicographic order. Libtool versioning counts distinct library interfaces.

Semantic versioning is written as `major.minor.patch` and libtool as `current:revision:age`. The intuition is that `current` counts interface changes. Any time the interface changes, whether in a minor or major way, `current` increases. Here's how each system would record the same history of release events:

Event	Semver	Libtool
Initial	1.0.0	1:0:0
Minor	1.1.0	2:0:1
Minor	1.2.0	3:0:2
Patch	1.2.1	3:1:2
Major	2.0.0	4:0:0
Patch	2.0.1	4:1:0
Patch	2.0.2	4:2:0
Minor	2.1.0	5:0:1

Here's how applications answer the question, **"Can I load a given library?"**

Semver
Does the library to be loaded have the same major version as the library I linked with, and a minor version at least as big?
Libtool
Is the `current` interface number of the library I linked with between `current - age` and `current` of the library to be loaded?

We'll be using semantic versioning in this guide, because libtool versioning is only relevant to libtool, a tool to abstract library creation across platforms. I believe we can make portable libraries without libtool. I mention both systems only to show that there's more than one way to build version identifiers.

One final note: version identifiers say that things *have* changed, but omit *what* changed. More complicated systems exist to track library compatibility. Solaris, for instance, developed a system called symbol versioning. Symbol versioning chases space savings at the expense of operational complexity, and we'll consider it later.

API vs ABI

One subtlety of versioning is that changes can happen in either a library's *programming* interface (API) or *binary* interface (ABI). A C library's programming interface is defined through its header files. A backward-incompatible API change means a program written for the previous version would not compile when including headers from the new version.

By contrast, a binary interface is a runtime concept. It concerns the calling conventions for functions, or the memory layout (and meaning) of data shared between program and library. The ABI ensures compatibility at load and run-time, while the API ensures compatibility at compile and link time.

The two interfaces usually change hand-in-hand, and people sometimes confuse them. It's possible for one to change without the other, though.

Examples of breaking ABI, but API stability:

In these library changes, application code doesn't need to change, but does need to be recompiled with the new library headers in order to work at runtime.

- A changed numerical value behind a #define constant. A program compiled before the change would pass the wrong value to the library.
- Reordered elements in a struct. The program and library would read different offsets in memory thinking they're referring to the same element, which is definitely an ABI break. Even adding an element *after* the others would affect the structure's size, and hence layout within an array. An added field at the end may or may not affect a particular library's ABI.
- Widening function arguments. For instance changing a short int argument to a long int on an architecture/compiler where their size differs. Recompilation would be necessary to handle e.g. sign extension, or the offset of the next argument.
- Other languages, including C++, have more opportunities for surprise ABI breakage.

Examples of ABI stability, but breaking API:

In these library changes, application code would need to be modified to compile successfully against the new library, even though code compiled before the change could load and call the library without issue.

- Changing an argument from `const foo *` to `foo *`. A pointer to a const object cannot be implicitly converted to a pointer to a non-const object. The ABI doesn't care though, and moves the same bytes. (If the library does in fact modify the dereferenced value, it may be an unpleasant surprise to the application of course.)
- Changing the name of a struct element, while keeping its meaning and leaving it in the same position relative to the other elements.

It's usually easy to tell when you've added functionality vs broken backward compatibility, but there are tools to check for sure. For instance, the ABI Compliance Checker (https://lvc.github.io/abi-compliance-checker/) can detect breakages in C and C++ libraries.

In light of the versioning discussion earlier, which changes should the version identifier describe? At the very least, the ABI. When the loader is searching for a library, the ABI determines whether a library would be compatible at runtime. However, I think a more conservative versioning scheme is wise, where you bump a version when *either* the API or ABI change. You'll end up with potentially more library versions installed, but each shared API/ABI version will provide guarantees at both compilation and runtime.

Variance of linker and loader by system

Linkers (ld, lld)

After compiling object files, the compiler front-end (gcc, clang, cc, c99) will invoke the linker (ld, lld) to find unresolved symbols and match them across object files or in shared libraries. The linker searches only the shared libraries requested by the

front-end, in the order specified on the command line. If an unresolved symbol is found in a listed library, the linker marks a dependency on that library in the generated executable.

The `-l` option adds a library to the list of candidates for symbol search. To add `libfoo.so` (or `libfoo.dylib` on Mac), specify `-lfoo`. The linker looks for the library files in its search path. To add directories to the default search path(s), use `-L`, for instance `-L/usr/local/lib`.

What happens if multiple versions of a library exist in the same directory? For instance two major versions, `libfoo.so.1` and `libfoo.so.2`? OpenBSD knows about version numbers, and would pick the highest version automatically for `-lfoo`. Linux and Mac would match neither, because they're looking for an exact match of `libfoo.so` (or `libfoo.dylib`). Similarly, what if both a static and dynamic library exist in the same directory, `libfoo.a` and `libfoo.so`? All systems will choose the dynamic one.

Greater control is necessary. GCC has a colon option to solve the problem, for instance `-l:libfoo.so.1`. However clang doesn't have it, so a truly portable build shouldn't rely on it. Some systems solve the problem by creating a symlink from `libfoo.so` to the specific library desired. However when done in a system location like `/usr/local/lib`, it nominates a single inflexible link-time version for the whole system. I'll suggest a different solution later that involves storing link-time files in a separate place from load-time libraries.

Loaders (ld.so, dyld)

At launch time, programs with dynamic library dependencies load and run ld.so (or dyld on Mac) to find and load the rest of their dependencies. The load library inspects DT_NEEDED ELF tags (or LOAD_DYLIB names in Mach-O on Mac) to determine which library filename to find on the system. Interestingly, these values are not specified by the program developer, but by the library developer. They are extracted from the libraries themselves at link-time.

Dynamic libraries contain an internal “runtime name” called SONAME in ELF, or install_name in Mach-O. An application may link against a file named `libfoo.so`, but the library SONAME can say, “search for me under the filename libfoo.so.1.2 at load time.” The loader cares only about filenames, it never consults SONAMES. Conversely, the linker’s output cares only about SONAMES, not input library filenames.

Loaders in different operating systems go about finding dependent libraries slightly differently. OpenBSD’s ld.so is very true to the SunOS model, and understands semantic versions (https://www.openbsd.org/faq/ports/specialtopics.html#SharedLibs). For instance, if asked to load libfoo.so.1.2, it will attempt to find libfoo.so.1.x with the largest x ≥ 2. FreeBSD also claims (https://docs.freebsd.org/en/books/developers-handbook/policies/#policies-shlib) to have this behavior, but I didn’t observe it in my tests (https://github.com/begriffs/test-ld.so).

In 1995, Solaris 2.5 created a way to track semantic versioning at the symbol level, rather than for the entire library. With symbol versioning there would be a single e.g. libfoo.so file that simply grows over time. Every function inside is marked with a version number. The same function name can even exist under multiple versions with different implementations.

The advantage of symbol versioning is that it can save space. In the alternative, where versioning is per-library rather than per-symbol, a large percentage of object code is often copied unchanged from one library version to the next. The disadvantages of symbol versioning are:

1. It’s harder to see exactly which versions are installed on a system. Versions are hidden within libraries, rather than visible in filenames.
2. Library developers have to maintain a separate symbol mapfile for the linker.

Symbol versioning quickly found its way into Linux, and became a staple of Glibc. Because of Linux’s symbol versioning preference, its ld.so doesn’t make any effort to rendezvous with the latest minor library version (à la SunOS or OpenBSD). Ld.so

searches for an exact match between SONAME and filename.

However, even on Linux, most libraries don't use symbol versioning. Also, their SONAMEs typically record only a major version (like libfoo.so.2). Within that major version, you just have to hope the hidden minor version is new enough for all applications compiled or installed on the system. If an app relies on functions added in a later minor library version, it'll crash when it attempts to call them. (Setting the environment variable `LD_BIND_NOW=1` will attempt to resolve all symbols at program start instead, to detect the failure up front.)

MacOS uses an entirely different object format (Mach-O rather than ELF), and a differently named loader library (dyld rather than ld.so). Mac's dynamically linked libraries are named `.dylib`, and their version numbers precede the extension.

Native Mac applications are usually installed into their own dedicated directories, with libraries bundled inside. Thus the loader has special provisions for finding libraries, like the keywords `@executable_path`, `@loader_path` and `@rpath` in the `install_name`. MacOS supports system libraries too, with dyld consulting the `DYLD_FALLBACK_LIBRARY_PATH`, by default `$(HOME)/lib:/usr/local/lib:/lib:/usr/lib`.

Like Linux, Mac does an exact name match – no minor version rendezvous. Unlike Linux, libraries can record their full semantic version internally, and a "compatibility" version. The compatibility version gets copied into an application at link time, and says the application requires at least that version at runtime.

For example, `libfoo.1.dylib` with full version 1.2.3 should have a compatibility version of 1.2.0 according to the rules of semantic versioning. An application linked against it would refuse to load libfoo with lesser minor version, like 1.1.5. At load time, the user would see a clear error:

```
dyld: Library not loaded: libfoo.1.dylib
  Referenced from: myapp
  Reason: Incompatible library version: myapp requires version 1.2.0 or later,
          but libfoo.1.dylib provides version 1.1.5
```

Portable best practices

Linking

Standard practice is to create symlinks libfoo.so -> libfoo.so.x -> libfoo.so.x.y.z in a shared system directory. The first link (without the version number) is for linking at build time. Problem is, it's pinned to one version. There's no portable way to select which version to link against when there are multiple versions installed.

Also, standard practice gives even less care to versioning header files. Sometimes whichever version was most recently installed overwrites them in /usr/local/include. Sometimes the headers are maintained only at the major version level, in /usr/local/include/libfoo-n.

To solve these problems, I suggest bundling all development (linking) library files together into a different directory structure per version. Since I advocated earlier that the "total" library version should be bumped whenever the API *or* ABI changes, the same version safely applies to headers and binaries.

First choose an installation PREFIX. If the system has an /opt directory, pick that, otherwise /usr/local. In this directory, add dynamic and/or static libraries, headers, man pages, and pkg-config files as desired:

```
$PREFIX/libfoo-dev.x.y.z
├── libfoo.pc
├── libfoo-static.pc
├── include
│   └── foo
│       ├── ...
│       └── ...
├── lib
│   ├── libfoo.so (or dylib or dll)
│   └── static
│       └── libfoo.a
└── man
    ├── ...
    └── ...
```

Linking against libfoo.x.y.z is easy. In a Makefile, set your flags like this:

```
CFLAGS  += -I/opt/libfoo-dev.x.y.z/include
LDFLAGS += -L/opt/libfoo-dev.x.y.z/lib
LDLIBS  += -lfoo

# an example suffix rule using the flags
.c:
    $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $< $(LDLIBS)
```

Version flexibility with pkg-config

Pkg-config (https://www.freedesktop.org/wiki/Software/pkg-config/) can allow an application to express a range of acceptable library versions, rather than hardcoding a specific one. In a configure script, we'll test for the library's presence and version, and output the flags to `config.mk`:

```
# supposing we require libfoo 1.x for x >= 1
pkg-config --print-errors 'libfoo >= 1.1, libfoo < 2.0'

# save flags to config.mk
cat > config.mk <<-EOF
    CFLAGS += $(pkg-config --cflags libfoo)
    LDFLAGS += $(pkg-config --libs-only-L libfoo)
    LDLIBS += $(pkg-config --libs-only-l libfoo)
EOF
```

Then our Makefile becomes:

```
include config.mk

.c:
    $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $< $(LDLIBS)
```

To choose a specific version of libfoo, we can add it to the pkg-config search path and run the configure script:

```
# make desired libfoo version visible to pkg-config
export PKG_CONFIG_PATH="/opt/libfoo-dev.x.y.z:$PKG_CONFIG_PATH"

./configure
make
```

To create pkg-config `.pc` files for a library, see Dan Nicholson's guide (https://people.freedesktop.org/~dbn/pkg-config-guide.html). In order to offer both a static and dynamic library, the best way I could imagine was to release separate files, `libfoo.pc` and `libfoo-static.pc` that differ in their `-L` flag. One uses `lib` and another `lib/static`. (Pkg-config's `--static` flag is a bit of a misnomer, and just passes items in `Libs.private` *in addition to* `Libs` in the build process.)

Loading

This section talks about installing dynamic libraries for system-wide loading. Libraries installed for this purpose are not meant to link with at compile time, but to load at runtime.

ELF installation (BSD/Linux)

ELF objects don't have much version metadata. SONAME is about it. That, combined with the lackluster behavior of loaders on some systems, means the traditional installation technique doesn't work too well.

Let's review the traditional way to install ELF libraries, and then a safer method I designed.

Traditional installation method

1. For version x.y.z, compile libfoo.so with SONAME libfoo.so.x
2. Copy libfoo.so to /usr/local/lib/libfoo.so.x.y.z
3. Create symlink libfoo.so.x -> libfoo.x.y.z

This way allows a sysadmin to see exactly which versions are installed, and to have multiple major versions installed at once. It doesn't allow multiple minor versions per major (although usually only the latest minor is needed), and more importantly doesn't offer protection against loading too old a minor version.

Safer installation method

1. For version x.y.z, compile libfoo.so with SONAME libfoo.so.x.y

```
# use compilation flags
-shared -Wl,-soname,libfoo.so.${MAJOR}.${MINOR}
```

2. Copy libfoo.so to /usr/local/lib/libfoo.so.x.y.z

3. Backfill minor version symlinks in DEST:

```
i=0
while [ $i -le "$MINOR" ]; do
    ln -fs "libfoo.so.$VER" "$DEST/libfoo.so.$MAJOR.$i"
    i=$((i+1))
done
```

At the cost of potentially a lot of minor version symlinks, this technique emulates the SunOS and OpenBSD behavior of minor version rendezvous. Also, because the SONAME has major.minor granularity, it will protect against loading too old a minor version.

(As an alternative to the symlinks, FreeBSD has libmap.conf (https://nixdoc.net/man-pages/FreeBSD/man5/libmap.conf.5.html))

Mach-O installation (MacOS)

Mach-O has more version metadata inside than ELF, so a traditional install works fine here.

1. For version x.y.z, compile libfoo.dylib with
    - install_name libfoo.x.dylib
    - current version x.y.z
    - compatibility version x.y

```
# use compilation flags
-dynamiclib -install_name "libfoo.${MAJOR}.dylib" \
            -current_version ${VER} \
            -compatibility_version ${MAJOR}.${MINOR}.0
```

2. Copy libfoo.dylib to /usr/local/lib/libfoo.x.dylib

It's important to set the compatibility version correctly so that Mac's dyld will prevent loading too old a minor version. To upgrade the library, overwrite libfoo.x.dylib with one of a later internal minor release.

Example code

For an example of how to build a library portably, and install it conveniently for the linker and loader, see begriffs/libderp (https://github.com/begriffs/libderp). It's my first shared library, where I tested the ideas for this article.