Editor: Paul F. Dubois, paul@pfdubois.com

THE INSIDE STORY ON SHARED LIBRARIES AND DYNAMIC LOADING

By David M. Beazley, Brian D. Ward, and Ian R. Cooke

TRADITIONALLY, DEVELOPERS HAVE BUILT SCIENTIFIC SOFTWARE AS STAND-ALONE APPLICATIONS WRITTEN IN A SINGLE LANGUAGE SUCH AS FORTRAN, C, OR C++. HOWEVER, MANY scientists are starting to build their applications as extensions to scripting language interpreters or component frameworks. This often involves shared libraries and dynamically loadable modules. However, the inner workings of shared libraries and dynamic loading are some of the least understood and most mysterious areas of software development.

In this installment of Scientific Programming, we tour the inner workings of linkers, shared libraries, and dynamically loadable extension modules. Rather than simply providing a tutorial on creating shared libraries on different platforms, we want to provide an overview of how shared libraries work and how to use them to build extensible systems. For illustration, we use a few examples in C/C++ using the gcc compiler on GNU-Linux-i386. However, the concepts generally apply to other programming languages and operating systems.

Compilers and object files

When you build a program, the compiler converts source files to object files. Each object file contains the machine code instructions corresponding to the statements and declarations in the source program. However, closer examination reveals that object files are broken into a collection of sections corresponding to different parts of the source program. For example, the C program

```
#include <stdio.h>
int x = 42;

int main() {
 printf("Hello World, x = %d\n", x);
}
```

produces an object file that contains a *text section* with the machine code instructions of the program, a *data section* with the global variable *x*, and a "read-only" section with the string literal `Hello World, x = %d\n`. Additionally, the object file contains a symbol table for all the identifiers that appear in the source code. An easy way to view the symbol table is with the Unix command `nm`—for example,

```
$ nm hello.o
00000000 T main
         U printf
00000000 D x
```

For symbols such as `x` and `main`, the symbol table simply contains an offset indicating the symbol's position relative to the beginning of its corresponding section (in this case, `main` is the first function in the text section, and `x` is the first variable in the data section). For other symbols such as `printf`, the symbol is marked as undefined, meaning that it was used but not defined in the source program.

Linkers and linking

To build an executable file, the linker (for example, `ld`) collects object files and libraries. The linker's primary function is to bind symbolic names to memory addresses. To do this, it first scans the object files and concatenates the object file sections to form one large file (the text sections of all object files are concatenated, the data sections are concatenated, and so on). Then, it makes a second pass on the resulting file to bind symbol names to real memory addresses. To complete the second pass, each object file contains a relocation list, which contains symbol names and offsets within the object file that must be patched. For example, the relocation list for the earlier example looks something like this:

```
$ objdump -r hello.o

hello.o: file format elf32-i386

RELOCATION RECORDS FOR [.text]:
OFFSET         TYPE              VALUE
0000000a       R_386_32          x
```

```
00000010    R_386_32            .rodata
00000015    R_386_PC32          printf
```

Static libraries

To improve modularity and reusability, programming libraries usually include commonly used functions. The traditional library is an archive (`.a` file), created like this:

```
$ ar cr libfoo.a foo.o bar.o spam.o...
```

The resulting `libfoo.a` file is known as a *static* library. An archive's structure is nothing more than a collection of raw object files strung together along with a table of contents for fast symbol access. (On older systems, it is sometimes necessary to manually construct the table of contents using a utility such as the Unix `ranlib` command.)

When a static library is included during program linking, the linker makes a pass through the library and adds all the code and data corresponding to symbols used in the source program. The linker ignores unreferenced library symbols and aborts with an error when it encounters a redefined symbol.

Many compilers provide a pragma for declaring certain symbols as weak.

An often-overlooked aspect of linking is that many compilers provide a pragma for declaring certain symbols as weak. For example, the following code declares a function that the linker will include only if it's not already defined elsewhere.

```
#pragma weak foo
/* Only included by linker if not already defined */
void foo() {
    ...
}
```

Alternatively, you can use the weak pragma to force the linker to ignore unresolved symbols. For example, if you write the program

```
#pragma weak debug
extern void debug(void);
void (*debugfunc)(void) = debug;
int main() {
    printf("Hello World\n");
    if (debugfunc) (*debugfunc)();
}
```

the program compiles and links whether or not `debug()` is actually defined in any object file. When the symbol remains undefined, the linker usually replaces its value with 0. So, this technique can be a useful way for a program to invoke optional code that does not require recompiling the entire application (contrast this to enabling optional features with a preprocessor macro).

Although static libraries are easy to create and use, they present a number of software maintenance and resource utilization problems. For example, when the linker includes a static library in a program, it copies data from the library to the target program. If patching the library is ever necessary, everything linked against that library must be rebuilt for the changes to take effect. Also, copying library contents into the target program wastes disk space and memory—especially for commonly used libraries such as the C library. For example, if every program on a Unix machine included its own copy of the C library, the size of these programs would increase dramatically. Moreover, with a large number of active programs, a considerable amount of system memory goes to storing these copies of library functions.

Shared libraries

To address the maintenance and resource problems with static libraries, most modern systems now use shared libraries or dynamic link libraries (DLLs). The primary difference between static and shared libraries is that using shared libraries delays the actual task of linking to runtime, where it is performed by a special dynamic linker–loader. So, a program and its libraries remain decoupled until the program actually runs.

Runtime linking allows easier library maintenance. For instance, if a bug appears in a common library, such as the C library, you can patch and update the library without recompiling or relinking any applications—they simply use the new library the next time they execute. A more subtle aspect of shared libraries is that they let the operating system make a number of significant memory optimizations. Specifically, because libraries mostly consist of executable instructions and this code is normally not self-modifying, the operating system can arrange to place library code in read-only memory regions shared among processes (using page-sharing and other virtual memory techniques). So, if hundreds of programs are running and each program includes the same library, the operating system can load a single shared copy of the library's instructions into physical memory. This reduces memory use and improves system performance.

Cafe Dubois

The Times, They Are a Changin'

Twenty years of schoolin' and they put you on the day shift.
—Bob Dylan

This summer marks my 25th year at Lawrence Livermore National Laboratory, all of it on the day shift. LLNL is a good place to work if you are someone like me who likes to try new areas, because you can do it without moving to a new company.

When my daughter was in the fifth grade, she came to Take Your Daughter to Work Day, and afterwards told me, referring to the system of community bicycles that you can ride around on, "The Lab is the greatest place in the world to work. They have free bikes and the food at the cafeteria is yummy!" After that day she paid a lot of attention to her math and science. Free bikes and yummy food is a lot of motivation. She's off to college this year, and I will miss her.

We technical types live in such a constant state of change, and it is so hard to take the time to keep up. For each of us, the time will come when we have learned our last new thing, when we tell ourselves something is not worth learning when the truth is we just can't take the pain anymore. So, when I decide not to learn something these days, I worry about my decision. Was that the one? Is it already too late?

Was it Java Beans? I sure hope it wasn't Java Beans. What an ignominious end that would be.

F90 pointers

In my article on Fortran 90's space provisions, I didn't have space to discuss pointers. One reader wrote me about having performance problems allocating and deallocating a lot of small objects. So, here is a simple "small object cache" module that will give you the idea of how to use pointers. In this module, one-dimensional objects of size *N* or smaller can be allocated by handing out columns of a fixed cache. The free slots are kept track of through a simple linked list. If the cache fills up, we go to the heap:

```
module soc
    ! Allocate memory of size <= N from a fixed block.
    private
    public get, release, init_soc
```

Paul in Paris, considering how life imitates art.

```
    integer, parameter:: N=16, M=100
    real, target:: cache(N, M)
    integer::links(M), first

contains

    subroutine init_soc ()
        integer i
        do i = 1, M-1
            links(i) = i + 1
        enddo
        links(M) = -1
        first = 1
    end subroutine init_soc

    function get(s)
        integer, intent(in):: s
        real, pointer:: get(:)
        integer k
        if (s > N) then
           allocate(get(s))
           return
        endif
        if (first == -1) then
            allocate(get(s))
```

On most systems, the static linker handles both static and shared libraries. For example, consider a simple program linked against a few different libraries:

```
$ gcc hello.c -lpthread -lm
```

If the libraries `-lpthread` and `-lm` have been compiled as shared libraries (usually indicated by a `.so` suffix on the actual library file), the static linker checks for unresolved symbols and reports errors as usual. However, rather than copying the contents of the libraries into the target executable, the linker simply records the names of the libraries in a list in the executable. You can view the contents of the library dependency list with a command such as `ldd`:

```
ldd a.out
```

```
            return
        endif
        k = first
        first = links(k)
        get => cache(1:s, k)
        return
    end function get

    subroutine release(x)
        real, pointer:: x(:)
        integer i
        if (size(x) > N) then
            deallocate(x)
            return
        endif
        do i = 1, M
           if (associated(x, cache(1:size(x), i))) then
               links(i) = first
               first = i
               return
           endif
        enddo
        deallocate(x)
    end subroutine release

end module soc

program socexample
    use soc
    real, pointer:: x1(:), x2(:), x3(:)
    integer i

    call init_soc ()
    x1 => get(3)
    x2 => get(3)
    x3 => get(20)

    x3 = (/ (i/2., i=1, 20) /)
    do i = 1, 3
       x1(i) = i
       x2(i) = -i
    enddo
    print *, x1+x2
    print *, x3
    call release(x2)
    call release(x1)
    call release(x3)
end program socexample
```

The input queue is low just now and I'd love to hear from authors about proposed articles. Just email me at paul@pfdubois.com. And remember, if it's Java Beans you want, it ain't me you're lookin' for, babe.

```
libpthread.so.0 => /lib/libpthread.so.0 (0x40017000)
libm.so.6 => /lib/libm.so.6 (0x40028000)
libc.so.6 => /lib/libc.so.6 (0x40044000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

When binding symbols at runtime, the dynamic linker searches libraries in the same order as they were specified on the link line and uses the first definition of the symbol encountered. If more than one library happens to define the same symbol, only the first definition applies. Duplicate symbols normally don't occur, because the static linker scans all the libraries and reports an error if duplicate symbols are defined. However, duplicate symbol names might exist if they are weakly defined, if an update to an existing shared library introduces new names that conflict with other libraries, or if a setting of the `LD_LIBRARY_PATH` variable subverts the load path (described later).

By default, many systems export all the globally defined symbols in a library (anything accessible by using an `extern` specifier in C/C++). However, on certain platforms, the list of exported symbols is more tightly controlled with export lists, special linker options, or compiler extensions. When these extensions are required, the dynamic linker will bind only to symbols that are explicitly exported. For example, on Windows, exported library symbols must be declared using compiler-specific code such as this:

```
__ declspec(dllexport) extern void foo(void);
```

An interesting aspect of shared libraries is that the linking process happens at each program invocation. To minimize this performance overhead, shared libraries use both *indirection tables* and *lazy symbol binding*. That is, the location of external symbols actually refers to table entries, which remain unbound until the application actually needs them. This reduces startup time because most applications use only a small subset of library functions.

To implement lazy symbol binding, the static linker creates a jump table known as a *procedure-linking table* and includes it as part of the final executable. Next, the linker resolves all unresolved function references by making them point directly to a specific PLT entry. So, executable programs created by the static linker have an internal structure similar to that in Figure 1. To make lazy symbol binding work at runtime, the dynamic linker simply clears all the PLT entries and sets them to point to a special symbol-binding function inside the dynamic library loader. The neat part about this trick is that as each library function is used for the first time, the dynamic linker regains control of the process and performs all the necessary symbol bindings. After it locates a symbol, the linker simply overwrites the corresponding PLT entry so that subsequent calls to the same function transfer control directly to the function instead of calling the dynamic linker again. Figure 2 illustrates an overview of this process.

Although symbol binding is normally transparent to users, you can watch it by setting the `LD_DEBUG` environment variable to the value bindings before starting your program.

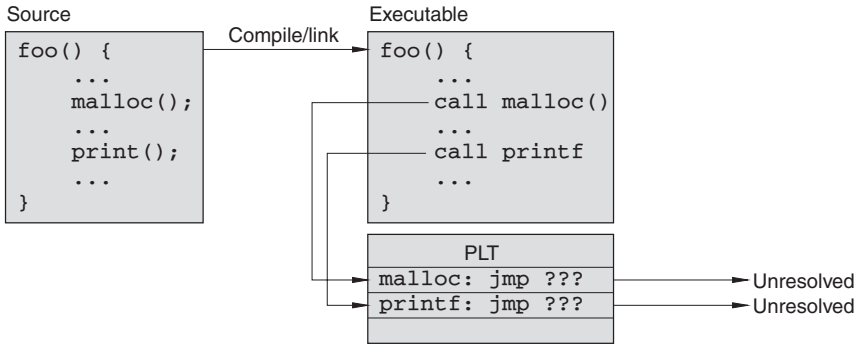

Figure 1. The internal structure of an executable linked with shared libraries. External library calls point to procedure-linking table entries, which remain unresolved until the dynamic linker fills them in at runtime.

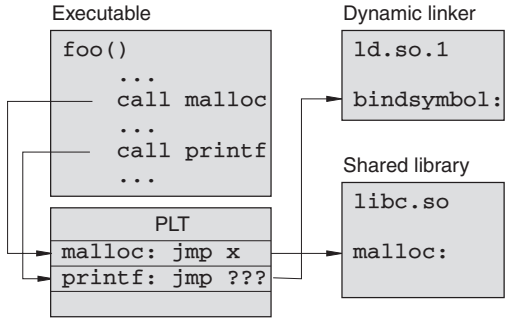

Figure 2. The dynamic binding of library symbols in shared libraries: `malloc` is bound to the C library, and `printf` has not yet been used and is bound to the dynamic linker.

An interesting experiment is to watch the dynamic symbol binding for interactive applications such as an interpreter or a browser—for example,

```
$ LD_DEBUG=bindings python
```

If you really enjoy copious debugging information, you can set `LD_DEBUG=bindings` in the shell and start running a few programs.

Library loading

When a program linked with shared libraries runs, program execution does not immediately start with that program's first statement. Instead, the operating system loads and executes the dynamic linker (usually called `ld.so`), which then scans the list of library names embedded in the executable. These library names are never encoded with absolute pathnames. Instead, the list has only simple names such as `libpthread.so.0`, `libm.so.6`, and `libc.so.6`, where the last digit is a library version number. To locate the libraries, the dynamic linker uses a configurable library search path. This path's default value is normally stored in a system configuration file such as `/etc/ld.so.conf`. Additionally, other library search directories might be embedded in the executable or specified by the user in the `LD_LIBRARY_PATH` environment variable.

Libraries always load in the order in which they were linked. Therefore, if you linked your program like this,

```
$ cc $SRCS -lfoo -lbar -lsocket -lm -lc
```

the dynamic linker loads the libraries in the order `libfoo.so`, `libbar.so`, `libsocket.so`, `libm.so`, and so forth. If a shared library includes additional library dependencies, those libraries are appended to the end of the library list during loading. For example, if the library `libbar.so` depends on an additional library `libspam.so`, that library loads after all the other libraries on the link line (such as those after `libc.so`). To be more precise, the library loading order is determined by a breadth-first traversal of library dependencies starting with the libraries directly linked to the executable. Internally, the loader keeps track of the libraries by placing them on a linked list known as a *linkchain*. Moreover, the loader guarantees that no library is ever loaded more than once (previously loaded copies are used when a library is repeated).

Because directory traversal is relatively slow, the loader does not look at the directories in `/etc/ld.so.conf` every time it runs to find library files, but consults a cache file instead. Normally named `/etc/ld.so.cache`, this cache file is a table that matches library names to full pathnames. If you add a new shared library to a library directory listed in `/etc/ld.so.conf`, you must rebuild the cache file with the `ldconfig` command, or the loader won't find it. The `-v` option produces verbose detail, including any new or altered libraries.

If the dynamic loader can't find a library in the cache, it often makes a last-ditch effort with a manual search of system library directories such as `/lib` and `/usr/lib` before it gives up and returns an error. This behavior depends on the operating system.

You can obtain detailed information about how the dynamic linker loads libraries by setting the `LD_DEBUG` environment variable to `libs`—for example,

```
$ LD_DEBUG=libs a.out
```

When users first work with shared libraries, they commonly experience error messages related to missing libraries. For example, if you create your own shared library such as this,

```
$ cc -shared $OBJS -o libfoo.so
```

and link an executable with the library, you might get the following error when you try to run your program:

```
./a.out: error in loading shared libraries: libfoo.so:
cannot open shared object file: No such file or directory
```

This error usually occurs when the shared library is placed in a nonstandard location (not defined in `ld.so.conf`). A common hack to fix this problem is to set the `LD_LIBRARY_PATH` environment variable. However, setting `LD_LIBRARY_PATH` is nearly always a bad idea. One common mistake is to set it in your default user environment. Because there is no cache for this user-defined variable, `ld.so` must search through each entry of every directory in this path for a library before it looks anywhere else. So, virtually every command you run makes `ld.so` do more work than it really should whenever it needs to access a shared library.

A more serious problem is that this approach invites library clashes. A classic example of this was in SunOS 4, which shipped with the default user environment's `LD_LIBRARY_PATH` set to `/usr/openwin/lib`. Because SunOS 4 had many incompatibilities with several packages, systems administrators often had to compile the MIT X distribution. However, users ended up with the libraries specified by the environment variables and more warnings about older versions of libraries than expected (and if they were lucky, had their programs crash).

A better solution to the library path problem is to embed customized search paths in the executable itself using special linker options such as `-R` or `-Wl`, `-rpath`—for example,

```
$ cc $SRCS -Wl,-rpath=/home/beazley/libs \
-L/home/beazleys/libs  -lfoo
```

In this case, the program will find `libfoo.so` in the proper directory without having to set any special environment variables. Even with this approach, managing shared libraries is tricky. If you simply put your custom shared libraries in a place such as `/usr/local/lib`, you might have problems if the API changes when you upgrade the library. On the other hand, if you put them somewhere else, a user might not be able to find them. Some library packages such as gtk+ come with a command that, with certain flags, spits back the linker options you need for shared libraries (you embed the command in your `configure` script or `Makefile`).

Library initialization and finalization

As libraries are loaded, the system must occasionally perform certain preliminary steps. For example, if a C++ program has any statically constructed objects, they must be initialized before program startup. For example,

```
class Foo {
    public:
          Foo();
          ~Foo();
    ...
}

/* Statically initialized object */
Foo f;
```

To handle this situation, the dynamic linker looks for a special `_init()` function or "init" section in each loaded library. The compiler creates the contents of `_init()` and contains the startup code needed to initialize objects and other parts of the runtime environment. The invocation of `_init()` functions follows in the reverse order of library loading (for example, the first library loaded is the last to call `_init()`). This reversed ordering is necessary because libraries appearing earlier on the link line might depend on functions used in libraries appearing later.

When libraries unload at program termination, the dynamic linker looks for special `_fini()` functions and invokes them in the opposite order as the `_init()` functions. The role of `_fini()` is to destroy objects and perform other kinds of cleanup.

If you're curious, you can also trace debugging information about the invocation of `_init()` and `_fini()` functions by setting the environment variable `LD_DEBUG` to `libs`.

Dynamic loading

So far, our discussion has focused primarily on the underlying implementation of shared libraries and how they are organized to support runtime linking of programs. An added feature of the dynamic linker is an API for accessing symbols and loading new libraries at runtime (*dynamic loading*). The dynamic loading mechanism is a critical part of many extensible systems, including scripting language interpreters.

Dynamic loading is usually managed by three functions exposed by the dynamic linker: `dlopen()` (which loads a new shared library), `dlsym()` (which looks up a specific symbol in the library), and `dlclose()` (which unloads the library and removes it from memory)—for example,

```
void *handle;
void (*foo)(void);
```

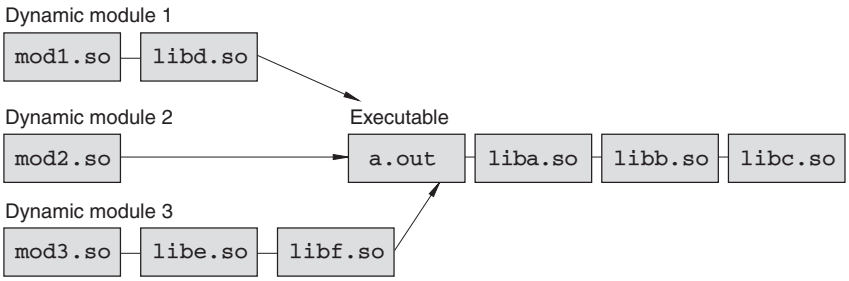

Figure 3. Linkchains created by dynamic loading. Each dynamically loadable module goes on a new linkchain but can still bind to symbols defined in the primary executable.

```
handle = dlopen("foo.so", RTLD_NOW | RTLD_LOCAL);
foo = (void (*)(void)) dlsym(handle,"foo");
if (foo) foo();
...
dlclose(handle);
```

Unlike standard linking, dynamically loaded modules are completely decoupled from the underlying application. The dynamic loader does not use them to resolve any unbound symbols or any other part of the normal application-linking process.

The dynamic linker loads a module and all of its required libraries using the exact same procedure as before (that is, it loads libraries in the order specified on the link line, `_init()` functions are invoked, and so forth). Again, the linker keeps track of previously loaded libraries and will not reload shared libraries that are already present. The main difference is that when modules are loaded, they normally go on private *linkchains*. This results in a tree of library dependencies (see Figure 3).

As symbols are bound in each dynamically loaded module, the linker searches libraries starting with the module itself and the list of libraries in its corresponding linkchain. Additionally, the linker searches for symbols in the main executable and all its libraries.

One consequence of the tree structure is that each dynamically loaded module gets its own private symbol namespace. So, if a symbol name is defined in more than one loadable module, those symbols remain distinct and do not clash with each other during execution. Similarly, unless the program gives special options to the dynamic loader, it doesn't use symbols defined in previously loaded modules to resolve symbols in newly loaded modules. For users of scripting languages, this explains why everything still works even when two extension modules appear to have a namespace clash. It also explains why extension modules can't dynamically bind to symbols defined in other dynamically loaded modules.

Filters

Dynamic linking enables modes of linking that are not easily achieved with static libraries. One such example is the implementation of filters that let you alter the behavior of common library functions. For example, suppose you want to track calls to dynamic memory-allocation functions `malloc()` and `free()`. One way to do this with dynamic linking is to write a simple library as in Figure 4. In the figure, we implemented replacements for the standard memory management functions. However, these replacements use `dlsym()` to search for the real implementation of `malloc()` and `free()` farther down the linkchain.

To use this filter, the library can be included on the linkline like a normal library. Alternatively, the library can be preloaded by supplying a special environment variable to the dynamic linker. The following command illustrates library preloading by running Python with our memory-allocation filter enabled:

```
$ LD_PRELOAD=./libmfilter.so python
```

Depending on how you tweak the linker, the user can place filters almost anywhere on the linkchain. For instance, if the filter functions were linked to a dynamically loadable module, only the memory allocations performed by the module pass through the filters; all other allocations remain unaffected.

Constructing shared libraries

Let's examine a few problematic aspects of shared library construction. First, when shared libraries and dynamically loadable modules are compiled, it is fairly common to see special compiler options for creating *position-independent code* (`-fpic`, `-FPIC`, `-KPIC`, and so on). When these options are present during compilation, the compiler generates code that accesses symbols through a collection of indirection registers and global offset tables. The primary benefit of PIC code is that the text segment of the resulting library (containing machine instructions) can quickly relocate to any memory address without code patches. This improves program startup time and is important for libraries that large numbers of running processes must share. (PIC lets the operating system relocate libraries to different virtual memory regions without modifying the library instructions.) The downside to PIC is a modest degradation in application performance (often 5 to 15 percent). A somewhat overlooked point in many shared library examples is that shared libraries and dynamically loadable modules generally do not require PIC code. So, if performance is important for a library or dynamically loadable module, you can compile it as non-PIC code. The primary downside to compiling the module as non-PIC is that loading time increases because the dynamic linker must make a large number of code patches when binding symbols.

A second problem with shared library construction concerns the inclusion of static libraries when building dynamically loadable modules. When the user or programmer works with

```
/* libmfilter.c */
#include <dlfcn.h>
#include <stdio.h>
#include <assert.h>

typedef void *(*malloc_t)(size_t nbytes);

void *malloc(size_t nbytes) {
    void *r;
    static malloc_t real_malloc = 0;
    if (!real_malloc) {
          real_malloc = (malloc_t) dlsym(RTLD_NEXT, "malloc");
          assert(real_malloc);
    }
    r = (*real_malloc)(nbytes);
    printf("malloc %d bytes at %x\n", nbytes, r);
    return r;
}

typedef void (*free_t)(void *ptr);
void free(void *ptr) {
    static free_t real_free = 0;
    if (!real_free) {
          real_free = (free_t) dlsym(RTLD_NEXT, "free");
          assert(real_free);
    }
    printf("free %x\n", ptr);
    (*real_free)(ptr);
}
```

Figure 4. Filters for `malloc` and `free`. The filters look up the real implementation using `dlsym()` and print debugging information.

scripting language interpreters and other extensible systems, it is common to create various types of modules for different parts of an application. However, if parts of the application are built as static libraries (`.a` files), the linking process does not do what you might expect. Specifically, when a static library is linked into a shared module, the relevant parts of the static library are simply copied into the resulting shared library object. For multiple modules linked in this manner, each module ends up with its own private copy of the static library. When these modules load, all private library copies remain isolated from each other, owing to the way in which symbols are bound in dynamic modules (as described earlier). The end result is grossly erratic program behavior. For instance, changes to global variables don't seem to affect other modules, functions seem to operate on different sets of data, and so on. To fix this problem, convert static libraries to shared libraries. Then, when multiple dynamic modules link again to the library, the dynamic linker will guarantee that only one copy of the library loads into memory.

A few final words and further reading

As extensible systems become more popular in scientific software, scientists must have a firm understanding of how shared libraries and dynamically loadable modules interact with each other and fit into the overall picture of large applications. We have described many of the general concepts and techniques used behind the scenes on these systems. Although our discussion focused primarily on Unix, high-level concepts apply to all systems that use shared libraries and dynamic linking.

You can find further information about linkers, loaders, and dynamic loading elsewhere.[1] Advanced texts on programming languages and compilers often contain general information about PLTs, lazy binding, and shared library implementation.[2] Several research papers on operating systems contain the specific implementation details of dynamic linking.[3,4] CiSE

References

1. J.R. Levine, *Linkers & Loaders*, Morgan Kaufmann, San Francisco, 2000.
2. M.L. Scott, *Programming Language Pragmatics*, Morgan Kaufmann, San Francisco, 2000.
3. R.A. Gingell et al., *Shared Libraries in SunOS*, Usenix, San Diego, Calif., 1987.
4. J.Q. Arnold, *Shared Libraries on UNIX System V*, Usenix, San Diego, Calif., 1986.

David Beazley is an assistant professor in the Department of Computer Science at the University of Chicago. He created the Simplified Wrapper and Interface Generator, a popular tool for creating scripting language extension modules, and wrote the *Python Essential Reference* (New Riders Publishing, 1999). Contact him at the Dept. of Computer Science, Univ. of Chicago, Chicago, IL 60637; beazley@cs.uchicago.edu.

Brian Ward is a PhD candidate in the Department of Computer Science at the University of Chicago. He wrote the Linux Kernel HOWTO, *The Linux Problem Solver* (No Starch Press, 2000), and *The Book of VMware* (No Starch Press, forthcoming). Contact him at the Dept. of Computer Science, Univ. of Chicago, Chicago, IL 60637; bri@cs.uchicago.edu.

Ian Cooke is a PhD student in the Department of Computer Science at the University of Chicago. Contact him at the Dept. of Computer Science, Univ. of Chicago, Chicago, IL 60637; iancooke@cs.uchicago.edu.