42

ADVANCED FEATURES OF SHARED LIBRARIES

The previous chapter covered the fundamentals of shared libraries. This chapter describes a number of advanced features of shared libraries, including the following:

- dynamically loading shared libraries;
- controlling the visibility of symbols defined by a shared library;
- using linker scripts to create versioned symbols;
- using initialization and finalization functions to automatically execute code when a library is loaded and unloaded;
- shared library preloading; and
- using `LD_DEBUG` to monitor the operation of the dynamic linker.

42.1 Dynamically Loaded Libraries

When an executable starts, the dynamic linker loads all of the shared libraries in the program's dynamic dependency list. Sometimes, however, it can be useful to load libraries at a later time. For example, a plug-in is loaded only when it is needed. This functionality is provided by an API to the dynamic linker. This API, usually referred to as the *dlopen* API, originated on Solaris, and much of it is now specified in SUSv3.

The *dlopen* API enables a program to open a shared library at run time, search for a function by name in that library, and then call the function. A shared library loaded at run time in this way is commonly referred to as a *dynamically loaded library*, and is created in the same way as any other shared library.

The core *dlopen* API consists of the following functions (all of which are specified in SUSv3):

- The *dlopen()* function opens a shared library, returning a handle used by subsequent calls.
- The *dlsym()* function searches a library for a symbol (a string containing the name of a function or variable) and returns its address.
- The *dlclose()* function closes a library previously opened by *dlopen()*.
- The *dlerror()* function returns an error-message string, and is used after a failure return from one of the preceding functions.

The *glibc* implementation also includes a number of related functions, some of which we describe below.

To build programs that use the *dlopen* API on Linux, we must specify the *-ldl* option, in order to link against the *libdl* library.

42.1.1 Opening a Shared Library: *dlopen()*

The *dlopen()* function loads the shared library named in *libfilename* into the calling process's virtual address space and increments the count of open references to the library.

```
#include <dlfcn.h>

void *dlopen(const char *libfilename, int flags);
                                Returns library handle on success, or NULL on error
```

If *libfilename* contains a slash (/), *dlopen()* interprets it as an absolute or relative pathname. Otherwise, the dynamic linker searches for the shared library using the rules described in Section 41.11.

On success, *dlopen()* returns a handle that can be used to refer to the library in subsequent calls to functions in the *dlopen* API. If an error occurred (e.g., the library couldn't be found), *dlopen()* returns NULL.

If the shared library specified by *libfilename* contains dependencies on other shared libraries, *dlopen()* also automatically loads those libraries. This procedure occurs recursively if necessary. We refer to the set of such loaded libraries as this library's *dependency tree*.

It is possible to call *dlopen()* multiple times on the same library file. The library is loaded into memory only once (by the initial call), and all calls return the same *handle* value. However, the *dlopen* API maintains a reference count for each library handle. This count is incremented by each call to *dlopen()* and decremented by each call to *dlclose()*; only when the count reaches 0 does *dlclose()* unload the library from memory.

The *flags* argument is a bit mask that must include exactly one of the constants RTLD_LAZY or RTLD_NOW, with the following meanings:

RTLD_LAZY
: Undefined function symbols in the library should be resolved only as the code is executed. If a piece of code requiring a particular symbol is not executed, that symbol is never resolved. Lazy resolution is performed only for function references; references to variables are always resolved immediately. Specifying the RTLD_LAZY flag provides behavior that corresponds to the normal operation of the dynamic linker when loading the shared libraries identified in an executable's dynamic dependency list.

RTLD_NOW
: All undefined symbols in the library should be immediately resolved before *dlopen()* completes, regardless of whether they will ever be required. As a consequence, opening the library is slower, but any potential undefined function symbol errors are detected immediately instead of at some later time. This can be useful when debugging an application, or simply to ensure that an application fails immediately on an unresolved symbol, rather than doing so only after executing for a long time.

  By setting the environment variable LD_BIND_NOW to a nonempty string, we can force the dynamic linker to immediately resolve all symbols (i.e., like RTLD_NOW) when loading the shared libraries identified in an executable's dynamic dependency list. This environment variable is effective in *glibc* 2.1.1 and later. Setting LD_BIND_NOW overrides the effect of the *dlopen()* RTLD_LAZY flag.

It is also possible to include further values in *flags*. The following flags are specified in SUSv3:

RTLD_GLOBAL
: Symbols in this library and its dependency tree are made available for resolving references in other libraries loaded by this process and also for lookups via *dlsym()*.

RTLD_LOCAL
: This is the converse of RTLD_GLOBAL and the default if neither constant is specified. It specifies that symbols in this library and its dependency tree are not available to resolve references in subsequently loaded libraries.

SUSv3 doesn't specify a default if neither RTLD_GLOBAL nor RTLD_LOCAL is specified. Most UNIX implementations assume the same default (RTLD_LOCAL) as Linux, but a few assume a default of RTLD_GLOBAL.

Linux also supports a number of flags that are not specified in SUSv3:

RTLD_NODELETE (since *glibc* 2.2)
: Don't unload the library during a *dlclose()*, even if the reference count falls to 0. This means that the library's static variables are not reinitialized if the library is later reloaded by *dlopen()*. (We can achieve a similar effect for libraries loaded automatically by the dynamic linker by specifying the *gcc −Wl,−znodelete* option when creating the library.)

RTLD_NOLOAD (since *glibc* 2.2)
: Don't load the library. This serves two purposes. First, we can use this flag to check if a particular library is currently loaded as part of the process's address space. If it is, *dlopen()* returns the library's handle; if it is not, *dlopen()* returns NULL. Second, we can use this flag to "promote" the *flags* of an already loaded library. For example, we can specify RTLD_NOLOAD | RTLD_GLOBAL in *flags* when using *dlopen()* on a library previously opened with RTLD_LOCAL.

RTLD_DEEPBIND (since *glibc* 2.3.4)
: When resolving symbol references made by this library, search for definitions in the library before searching for definitions in libraries that have already been loaded. This allows a library to be self-contained, using its own symbol definitions in preference to global symbols with the same name defined in other shared libraries that have already been loaded. (This is similar to the effect of the *−Bsymbolic* linker option described in Section 41.12.)

The RTLD_NODELETE and RTLD_NOLOAD flags are also implemented in the Solaris *dlopen* API, but are available on few other UNIX implementations. The RTLD_DEEPBIND flag is Linux-specific.

As a special case, we can specify *libfilename* as NULL. This causes *dlopen()* to return a handle for the main program. (SUSv3 refers to this as a handle for the "global symbol object.") Specifying this handle in a subsequent call to *dlsym()* causes the requested symbol to be sought in the main program, followed by all shared libraries loaded at program startup, and then all libraries dynamically loaded with the RTLD_GLOBAL flag.

42.1.2 Diagnosing Errors: *dlerror()*

If we receive an error return from *dlopen()* or one of the other functions in the *dlopen* API, we can use *dlerror()* to obtain a pointer to a string that indicates the cause of the error.

```
#include <dlfcn.h>

const char *dlerror(void);
```

Returns pointer to error-diagnostic string, or NULL if no error has occurred since previous call to *dlerror()*

The *dlerror()* function returns NULL if no error has occurred since the last call to *dlerror()*. We'll see how this is useful in the next section.

42.1.3 Obtaining the Address of a Symbol: *dlsym()*

The *dlsym()* function searches for the named *symbol* (a function or variable) in the library referred to by *handle* and in the libraries in that library's dependency tree.

```
#include <dlfcn.h>

void *dlsym(void *handle, char *symbol);
                                Returns address of symbol, or NULL if symbol is not found
```

If *symbol* is found, *dlsym()* returns its address; otherwise, *dlsym()* returns NULL. The *handle* argument is normally a library handle returned by a previous call to *dlopen()*. Alternatively, it may be one of the so-called pseudohandles described below.

> A related function, *dlvsym(handle, symbol, version)*, is similar to *dlsym()*, but can be used to search a symbol-versioned library for a symbol definition whose version matches the string specified in *version*. (We describe symbol versioning in Section 42.3.2.) The _GNU_SOURCE feature test macro must be defined in order to obtain the declaration of this function from <dlfcn.h>.

The value of a symbol returned by *dlsym()* may be NULL, which is indistinguishable from the "symbol not found" return. In order to differentiate the two possibilities, we must call *dlerror()* beforehand (to make sure that any previously held error string is cleared) and then if, after the call to *dlsym()*, *dlerror()* returns a non-NULL value, we know that an error occurred.

If *symbol* is the name of a variable, then we can assign the return value of *dlsym()* to an appropriate pointer type, and obtain the value of the variable by dereferencing the pointer:

```
int *ip;

ip = (int *) dlsym(symbol, "myvar");
if (ip != NULL)
    printf("Value is %d\n", *ip);
```

If *symbol* is the name of a function, then the pointer returned by *dlsym()* can be used to call the function. We can store the value returned by *dlsym()* in a pointer of the appropriate type, such as the following:

```
int (*funcp)(int);              /* Pointer to a function taking an integer
                                   argument and returning an integer */
```

However, we can't simply assign the result of *dlsym()* to such a pointer, as in the following example:

```
funcp = dlsym(handle, symbol);
```

The reason is that the C99 standard forbids assignment between a function pointer and *void **. The solution is to use the following (somewhat clumsy) cast:

```
*(void **) (&funcp) = dlsym(handle, symbol);
```

Having used *dlsym()* to obtain a pointer to the function, we can then call the function using the usual C syntax for dereferencing function pointers:

```
res = (*funcp)(somearg);
```

Instead of the **(void **)* syntax shown above, one might consider using the following seemingly equivalent code when assigning the return value of *dlsym()*:

```
(void *) funcp = dlsym(handle, symbol);
```

However, for this code, *gcc –pedantic* warns that "ANSI C forbids the use of cast expressions as lvalues." The **(void **)* syntax doesn't incur this warning because we are assigning to an address *pointed to* by the assignment's lvalue.

On many UNIX implementations, we can use casts such as the following to eliminate warnings from the C compiler:

```
funcp = (int (*) (int)) dlsym(handle, symbol);
```

However, the specification of *dlsym()* in SUSv3 *Technical Corrigendum Number 1* notes that the C99 standard nevertheless requires compilers to generate a warning for such a conversion, and proposes the **(void **)* syntax shown above.

> SUSv3 TC1 noted that because of the need for the **(void **)* syntax, a future version of the standard may define separate *dlsym()*-like APIs for handling data and function pointers. However, SUSv4 contains no changes with respect to this point.

Using library pseudohandles with *dlsym()*

Instead of specifying a library handle returned by a call to *dlopen()*, either of the following *pseudohandles* may be specified as the *handle* argument for *dlsym()*:

RTLD_DEFAULT
: Search for *symbol* starting with the main program, and then proceeding in order through the list of all shared libraries loaded, including those libraries dynamically loaded by *dlopen()* with the RTLD_GLOBAL flag. This corresponds to the default search model employed by the dynamic linker.

RTLD_NEXT
: Search for *symbol* in shared libraries loaded after the one invoking *dlsym()*. This is useful when creating a wrapper function with the same name as a function defined elsewhere. For example, in our main program, we may define our own version of *malloc()* (which perhaps does some bookkeeping of memory allocation), and this function can invoke the real *malloc()* by first obtaining its address via the call *func = dlsym(RTLD_NEXT, "malloc")*.

The pseudohandle values listed above are not required by SUSv3 (which nevertheless reserves them for future use), and are not available on all UNIX implementations. In order to get the definitions of these constants from <dlfcn.h>, we must define the _GNU_SOURCE feature test macro.

Example program

Listing 42-1 demonstrates the use of the *dlopen* API. This program takes two command-line arguments: the name of a shared library to load and the name of a function to execute within that library. The following examples demonstrate the use of this program:

```
$ ./dynload ./libdemo.so.1 x1
Called mod1-x1
```

```
$ LD_LIBRARY_PATH=. ./dynload libdemo.so.1 x1
Called mod1-x1
```

In the first of the above commands, *dlopen()* notes that the library path includes a slash and thus interprets it as a relative pathname (in this case, to a library in the current working directory). In the second command, we specify a library search path in LD_LIBRARY_PATH. This search path is interpreted according to the usual rules of the dynamic linker (in this case, likewise to find the library in the current working directory).

Listing 42-1: Using the *dlopen* API

shlibs/dynload.c

```
#include <dlfcn.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    void *libHandle;            /* Handle for shared library */
    void (*funcp)(void);        /* Pointer to function with no arguments */
    const char *err;

    if (argc != 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s lib-path func-name\n", argv[0]);

    /* Load the shared library and get a handle for later use */

    libHandle = dlopen(argv[1], RTLD_LAZY);
    if (libHandle == NULL)
        fatal("dlopen: %s", dlerror());

    /* Search library for symbol named in argv[2] */

    (void) dlerror();                           /* Clear dlerror() */
    *(void **) (&funcp) = dlsym(libHandle, argv[2]);
    err = dlerror();
    if (err != NULL)
        fatal("dlsym: %s", err);

    /* If the address returned by dlsym() is non-NULL, try calling it
       as a function that takes no arguments */

    if (funcp == NULL)
        printf("%s is NULL\n", argv[2]);
    else
        (*funcp)();

    dlclose(libHandle);                         /* Close the library */

    exit(EXIT_SUCCESS);
}
```

shlibs/dynload.c

42.1.4 Closing a Shared Library: *dlclose()*

The *dlclose()* function closes a library.

```
#include <dlfcn.h>

int dlclose(void *handle);
```

Returns 0 on success, or −1 on error

The *dlclose()* function decrements the system's counter of open references to the library referred to by *handle*. If this reference count falls to 0, and no symbols in the library are required by other libraries, then the library is unloaded. This procedure is also (recursively) performed for the libraries in this library's dependency tree. An implicit *dlclose()* of all libraries is performed on process termination.

> From *glibc* 2.2.3 onward, a function within a shared library can use *atexit()* (or *on_exit()*) to establish a function that is called automatically when the library is unloaded.

42.1.5 Obtaining Information About Loaded Symbols: *dladdr()*

Given an address in *addr* (typically, one obtained by an earlier call to *dlsym()*), *dladdr()* returns a structure containing information about that address.

```
#define _GNU_SOURCE
#include <dlfcn.h>

int dladdr(const void *addr, Dl_info *info);
```

Returns nonzero value if *addr* was found in a shared library, otherwise 0

The *info* argument is a pointer to a caller-allocated structure that has the following form:

```
typedef struct {
    const char *dli_fname;          /* Pathname of shared library
                                       containing 'addr' */
    void       *dli_fbase;          /* Base address at which shared
                                       library is loaded */
    const char *dli_sname;          /* Name of nearest run-time symbol
                                       with an address <= 'addr' */
    void       *dli_saddr;          /* Actual value of the symbol
                                       returned in 'dli_sname' */
} Dl_info;
```

The first two fields of the *Dl_info* structure specify the pathname and run-time base address of the shared library containing the address specified in *addr*. The last two fields return information about that address. Assuming that *addr* points to the exact address of a symbol in the shared library, then *dli_saddr* returns the same value as was passed in *addr*.

SUSv3 doesn't specify *dladdr()*, and this function is not available on all UNIX implementations.

42.1.6 Accessing Symbols in the Main Program

Suppose that we use *dlopen()* to dynamically load a shared library, use *dlsym()* to obtain the address of a function *x()* from that library, and then call *x()*. If *x()* in turn calls a function *y()*, then *y()* would normally be sought in one of the shared libraries loaded by the program.

Sometimes, it is desirable instead to have *x()* invoke an implementation of *y()* in the main program. (This is similar to a callback mechanism.) In order to do this, we must make the (global-scope) symbols in the main program available to the dynamic linker, by linking the program using the *--export-dynamic* linker option:

```
$ gcc -Wl,--export-dynamic main.c      (plus further options and arguments)
```

Equivalently, we can write the following:

```
$ gcc -export-dynamic main.c
```

Using either of these options allows a dynamically loaded library to access global symbols in the main program.

> The *gcc -rdynamic* option and the *gcc -Wl,-E* option are further synonyms for *-Wl,--export-dynamic*.

42.2 Controlling Symbol Visibility

A well-designed shared library should make visible only those symbols (functions and variables) that form part of its specified application binary interface (ABI). The reasons for this are as follows:

- If the shared library designer accidentally exports unspecified interfaces, then authors of applications that use the library may choose to employ these interfaces. This creates a compatibility problem for future upgrades of the shared library. The library developer expects to be able to change or remove any interfaces other than those in the documented ABI, while the library user expects to continue using the same interfaces (with the same semantics) that they currently employ.
- During run-time symbol resolution, any symbols that are exported by a shared library might interpose definitions that are provided in other shared libraries (Section 41.12).
- Exporting unnecessary symbols increases the size of the dynamic symbol table that must be loaded at run time.

All of these problems can be minimized or avoided altogether if the library designer ensures that only the symbols required by the library's specified ABI are exported. The following techniques can be used to control the export of symbols:

- In a C program, we can use the `static` keyword to make a symbol private to a source-code module, thus rendering it unavailable for binding by other object files.

As well as making a symbol private to a source-code module, the `static` keyword also has a converse effect. If a symbol is marked as `static`, then all references to the symbol in the same source file will be bound to that definition of the symbol. Consequently, these references won't be subject to run-time interposition by definitions from other shared libraries (in the manner described in Section 41.12). This effect of the `static` keyword is similar to the *–Bsymbolic* linker option described in Section 41.12, with the difference that the `static` keyword affects a single symbol within a single source file.

- The GNU C complier, *gcc*, provides a compiler-specific attribute declaration that performs a similar task to the `static` keyword:

```
void
__attribute__ ((visibility("hidden")))
func(void) {
    /* Code */
}
```

  Whereas the `static` keyword limits the visibility of a symbol to a single source code file, the `hidden` attribute makes the symbol available across all source code files that compose the shared library, but prevents it from being visible outside the library.

  As with the `static` keyword, the `hidden` attribute also has the converse effect of preventing symbol interposition at run time.

- Version scripts (Section 42.3) can be used to precisely control symbol visibility and to select the version of a symbol to which a reference is bound.
- When dynamically loading a shared library (Section 42.1.1), the *dlopen()* `RTLD_GLOBAL` flag can be used to specify that the symbols defined by the library should be made available for binding by subsequently loaded libraries, and the *––export–dynamic* linker option (Section 42.1.6) can be used to make the global symbols of the main program available to dynamically loaded libraries.

For further details on the topic of symbol visibility, see [Drepper, 2004 (b)].

42.3 Linker Version Scripts

A *version script* is a text file containing instructions for the linker, *ld*. In order to use a version script, we must specify the *––version–script* linker option:

```
$ gcc -Wl,--version-script,myscriptfile.map ...
```

Version scripts are commonly (but not universally) identified using the extension `.map`.

The following sections describe some uses of version scripts.

42.3.1 Controlling Symbol Visibility with Version Scripts

One use of version scripts is to control the visibility of symbols that might otherwise accidentally be made global (i.e., visible to applications linking against the library). As a simple example, suppose that we are building a shared library from

the three source files `vis_comm.c`, `vis_f1.c`, and `vis_f2.c`, which respectively define the functions *vis_comm()*, *vis_f1()*, and *vis_f2()*. The *vis_comm()* function is called by *vis_f1()* and *vis_f2()*, but is not intended for direct use by applications linked against the library. Suppose we build the shared library in the usual way:

```
$ gcc -g -c -fPIC -Wall vis_comm.c vis_f1.c vis_f2.c
$ gcc -g -shared -o vis.so vis_comm.o vis_f1.o vis_f2.o
```

If we use the following *readelf* command to list the dynamic symbols exported by the library, we see the following:

```
$ readelf --syms --use-dynamic vis.so | grep vis_
   30  12: 00000790    59    FUNC GLOBAL DEFAULT  10 vis_f1
   25  13: 000007d0    73    FUNC GLOBAL DEFAULT  10 vis_f2
   27  16: 00000770    20    FUNC GLOBAL DEFAULT  10 vis_comm
```

This shared library exported three symbols: *vis_comm()*, *vis_f1()*, and *vis_f2()*. However, we would like to ensure that only the symbols *vis_f1()* and *vis_f2()* are exported by the library. We can achieve this result using the following version script:

```
$ cat vis.map
VER_1 {
    global:
        vis_f1;
        vis_f2;
    local:
        *;
};
```

The identifier *VER_1* is an example of a *version tag*. As we'll see in the discussion of symbol versioning in Section 42.3.2, a version script may contain multiple *version nodes*, each grouped within braces ({}) and prefixed with a unique version tag. If we are using a version script only for the purpose of controlling symbol visibility, then the version tag is redundant; nevertheless, older versions of *ld* required it. Modern versions of *ld* allow the version tag to be omitted; in this case, the version node is said to have an anonymous version tag, and no other version nodes may be present in the script.

Within the version node, the `global` keyword begins a semicolon-separated list of symbols that are made visible outside the library. The `local` keyword begins a list of symbols that are to be hidden from the outside world. The asterisk (*) here illustrates the fact that we can use wildcard patterns in these symbol specifications. The wildcard characters are the same as those used for shell filename matching—for example, * and ?. (See the *glob(7)* manual page for further details.) In this example, using an asterisk for the `local` specification says that everything that wasn't explicitly declared `global` is hidden. If we did not say this, then *vis_comm()* would still be visible, since the default is to make C global symbols visible outside the shared library.

We can then build our shared library using the version script as follows:

```
$ gcc -g -c -fPIC -Wall vis_comm.c vis_f1.c vis_f2.c
$ gcc -g -shared -o vis.so vis_comm.o vis_f1.o vis_f2.o \
       -Wl,--version-script,vis.map
```

Using *readelf* once more shows that *vis_comm()* is no longer externally visible:

```
$ readelf --syms --use-dynamic vis.so | grep vis_
   25   0: 00000730    73    FUNC GLOBAL DEFAULT  11 vis_f2
   29  16: 000006f0    59    FUNC GLOBAL DEFAULT  11 vis_f1
```

42.3.2 Symbol Versioning

Symbol versioning allows a single shared library to provide multiple versions of the same function. Each program uses the version of the function that was current when the program was (statically) linked against the shared library. As a result, we can make an incompatible change to a shared library without needing to increase the library's major version number. Carried to an extreme, symbol versioning can replace the traditional shared library major and minor versioning scheme. Symbol versioning is used in this manner in *glibc* 2.1 and later, so that all versions of *glibc* from 2.0 onward are supported within a single major library version (`libc.so.6`).

We demonstrate the use of symbol versioning with a simple example. We begin by creating the first version of a shared library using a version script:

```
$ cat sv_lib_v1.c
#include <stdio.h>

void xyz(void) { printf("v1 xyz\n"); }
$ cat sv_v1.map
VER_1 {
        global: xyz;
        local:  *;      # Hide all other symbols
};
$ gcc -g -c -fPIC -Wall sv_lib_v1.c
$ gcc -g -shared -o libsv.so sv_lib_v1.o -Wl,--version-script,sv_v1.map
```

Within a version script, the hash character (#) starts a comment.

(To keep the example simple, we avoid the use of explicit library sonames and library major version numbers.)

At this stage, our version script, `sv_v1.map`, serves only to control the visibility of the shared library's symbols; *xyz()* is exported, but all other symbols (of which there are none in this small example) are hidden. Next, we create a program, *p1*, which makes use of this library:

```
$ cat sv_prog.c
#include <stdlib.h>

int
main(int argc, char *argv[])
{
    void xyz(void);

    xyz();

    exit(EXIT_SUCCESS);
}
$ gcc -g -o p1 sv_prog.c libsv.so
```

When we run this program, we see the expected result:

```
$ LD_LIBRARY_PATH=. ./p1
v1 xyz
```

Now, suppose that we want to modify the definition of *xyz()* within our library, while still ensuring that program *p1* continues to use the old version of this function. To do this, we must define two versions of *xyz()* within our library:

```
$ cat sv_lib_v2.c
#include <stdio.h>

__asm__(".symver xyz_old,xyz@VER_1");
__asm__(".symver xyz_new,xyz@@VER_2");

void xyz_old(void) { printf("v1 xyz\n"); }

void xyz_new(void) { printf("v2 xyz\n"); }

void pqr(void) { printf("v2 pqr\n"); }
```

Our two versions of *xyz()* are provided by the functions *xyz_old()* and *xyz_new()*. The *xyz_old()* function corresponds to our original definition of *xyz()*, which is the one that should continue to be used by program *p1*. The *xyz_new()* function provides the definition of *xyz()* to be used by programs linking against the new version of the library.

The two `.symver` assembler directives are the glue that ties these two functions to different version tags in the modified version script (shown in a moment) that we use to create the new version of the shared library. The first of these directives says that *xyz_old()* is the implementation of *xyz()* to be used for applications linked against version tag *VER_1* (i.e., program *p1* in our example), and that *xyz_new()* is the implementation of *xyz()* to be used by applications linked against version tag *VER_2*.

The use of `@@` rather than `@` in the second `.symver` directive indicates that this is the default definition of *xyz()* to which applications should bind when statically linked against this shared library. Exactly one of the `.symver` directives for a symbol should be marked using `@@`.

The corresponding version script for our modified library is as follows:

```
$ cat sv_v2.map
VER_1 {
        global: xyz;
        local:  *;      # Hide all other symbols
};

VER_2 {
        global: pqr;
} VER_1;
```

This version script provides a new version tag, *VER_2*, which depends on the tag *VER_1*. This dependency is indicated by the following line:

```
} VER_1;
```

Version tag dependencies indicate the relationships between successive library versions. Semantically, the only effect of version tag dependencies on Linux is that a version node inherits `global` and `local` specifications from the version node upon which it depends.

Dependencies can be chained, so that we could have another version node tagged *VER_3*, which depended on *VER_2*, and so on.

The version tag names have no meanings in themselves. Their relationship with one another is determined only by the specified version dependencies, and we chose the names *VER_1* and *VER_2* merely to be suggestive of these relationships. To assist maintenance, recommended practice is to use version tags that include the package name and a version number. For example, *glibc* uses version tags with names such as *GLIBC_2.0*, *GLIBC_2.1*, and so on.

The *VER_2* version tag also specifies that the new function *pqr()* is to be exported by the library and bound to the *VER_2* version tag. If we didn't declare *pqr()* in this manner, then the `local` specification that *VER_2* version tag inherited from the *VER_1* version tag would make *pqr()* invisible outside the library. Note also that if we omitted the `local` specification altogether, then the symbols *xyz_old()* and *xyz_new()* would also be exported by the library (which is typically not what we want).

We now build the new version of our library in the usual way:

```
$ gcc -g -c -fPIC -Wall sv_lib_v2.c
$ gcc -g -shared -o libsv.so sv_lib_v2.o -Wl,--version-script,sv_v2.map
```

Now we can create a new program, *p2*, which uses the new definition of *xyz()*, while program *p1* uses the old version of *xyz()*.

```
$ gcc -g -o p2 sv_prog.c libsv.so
$ LD_LIBRARY_PATH=. ./p2
v2 xyz                                          Uses xyz@VER_2
$ LD_LIBRARY_PATH=. ./p1
v1 xyz                                          Uses xyz@VER_1
```

The version tag dependencies of an executable are recorded at static link time. We can use *objdump −t* to display the symbol tables of each executable, thus showing the different version tag dependencies of each program:

```
$ objdump -t p1 | grep xyz
08048380       F *UND*  00000026              xyz@@VER_1
$ objdump -t p2 | grep xyz
080483a0       F *UND*  00000026              xyz@@VER_2
```

We can also use *readelf −s* to obtain similar information.

> Further information about symbol versioning can be found using the command *info ld scripts version* and at *http://people.redhat.com/drepper/symbol-versioning*.

42.4 Initialization and Finalization Functions

It is possible to define one or more functions that are executed automatically when a shared library is loaded and unloaded. This allows us to perform initialization and finalization actions when working with shared libraries. Initialization and finalization

functions are executed regardless of whether the library is loaded automatically or loaded explicitly using the *dlopen* interface (Section 42.1).

Initialization and finalization functions are defined using the *gcc* `constructor` and `destructor` attributes. Each function that is to be executed when the library is loaded should be defined as follows:

```
void __attribute__ ((constructor)) some_name_load(void)
{
    /* Initialization code */
}
```

Unload functions are similarly defined:

```
void __attribute__ ((destructor)) some_name_unload(void)
{
    /* Finalization code */
}
```

The function names *some_name_load()* and *some_name_unload()* can be replaced by any desired names.

> It is also possible to use the *gcc* `constructor` and `destructor` attributes to create initialization and finalization functions in a main program.

The *_init()* and *_fini()* functions

An older technique for shared library initialization and finalization is to create two functions, *_init()* and *_fini()*, as part of the library. The *void _init(void)* function contains code that is to executed when the library is first loaded by a process. The *void _fini(void)* function contains code that is to be executed when the library is unloaded.

If we create *_init()* and *_fini()* functions, then we must specify the *gcc −nostartfiles* option when building the shared library, in order to prevent the linker from including default versions of these functions. (Using the *−Wl,−init* and *−Wl,−fini* linker options, we can choose alternative names for these two functions if desired.)

Use of *_init()* and *_fini()* is now considered obsolete in favor of the *gcc* `constructor` and `destructor` attributes, which, among other advantages, allow us to define multiple initialization and finalization functions.

42.5 Preloading Shared Libraries

For testing purposes, it can sometimes be useful to selectively override functions (and other symbols) that would normally be found by the dynamic linker using the rules described in Section 41.11. To do this, we can define the environment variable `LD_PRELOAD` as a string consisting of space-separated or colon-separated names of shared libraries that should be loaded before any other shared libraries. Since these libraries are loaded first, any functions they define will automatically be used if required by the executable, thus overriding any other functions of the same name that the dynamic linker would otherwise have searched for. For example,

suppose that we have a program that calls functions *x1()* and *x2()*, defined in our *libdemo* library. When we run this program, we see the following output:

```
$ ./prog
Called mod1-x1 DEMO
Called mod2-x2 DEMO
```

(In this example, we assume that the shared library is in one of the standard directories, and thus we don't need to use the LD_LIBRARY_PATH environment variable.)

We could selectively override the function *x1()* by creating another shared library, libalt.so, which contains a different definition of *x1()*. Preloading this library when running the program would result in the following:

```
$ LD_PRELOAD=libalt.so ./prog
Called mod1-x1 ALT
Called mod2-x2 DEMO
```

Here, we see that the version of *x1()* defined in libalt.so is invoked, but that the call to *x2()*, for which no definition is provided in libalt.so, results in the invocation of the *x2()* function defined in libdemo.so.

The LD_PRELOAD environment variable controls preloading on a per-process basis. Alternatively, the file /etc/ld.so.preload, which lists libraries separated by white space, can be used to perform the same task on a system-wide basis. (Libraries specified by LD_PRELOAD are loaded before those specified in /etc/ld.so.preload.)

For security reasons, set-user-ID and set-group-ID programs ignore LD_PRELOAD.

42.6 Monitoring the Dynamic Linker: LD_DEBUG

Sometimes, it is useful to monitor the operation of the dynamic linker in order to know, for example, where it is searching for libraries. We can use the LD_DEBUG environment variable to do this. By setting this variable to one (or more) of a set of standard keywords, we can obtain various kinds of tracing information from the dynamic linker.

If we assign the value *help* to LD_DEBUG, the dynamic linker displays help information about LD_DEBUG, and the specified command is *not* executed:

```
$ LD_DEBUG=help date
Valid options for the LD_DEBUG environment variable are:

  libs       display library search paths
  reloc      display relocation processing
  files      display progress for input file
  symbols    display symbol table processing
  bindings   display information about symbol binding
  versions   display version dependencies
  all        all previous options combined
  statistics display relocation statistics
  unused     determine unused DSOs
  help       display this help message and exit

To direct the debugging output into a file instead of standard output
a filename can be specified using the LD_DEBUG_OUTPUT environment variable.
```

The following example shows an abridged version of the output provided when we request tracing of information about library searches:

```
$ LD_DEBUG=libs date
     10687:     find library=librt.so.1 [0]; searching
     10687:      search cache=/etc/ld.so.cache
     10687:       trying file=/lib/librt.so.1
     10687:     find library=libc.so.6 [0]; searching
     10687:      search cache=/etc/ld.so.cache
     10687:       trying file=/lib/libc.so.6
     10687:     find library=libpthread.so.0 [0]; searching
     10687:      search cache=/etc/ld.so.cache
     10687:       trying file=/lib/libpthread.so.0
     10687:     calling init: /lib/libpthread.so.0
     10687:     calling init: /lib/libc.so.6
     10687:     calling init: /lib/librt.so.1
     10687:     initialize program: date
     10687:     transferring control: date
Tue Dec 28 17:26:56 CEST 2010
     10687:     calling fini: date [0]
     10687:     calling fini: /lib/librt.so.1 [0]
     10687:     calling fini: /lib/libpthread.so.0 [0]
     10687:     calling fini: /lib/libc.so.6 [0]
```

The value 10687 displayed at the start of each line is the process ID of the process being traced. This is useful if we are monitoring several processes (e.g., parent and child).

By default, LD_DEBUG output is written to standard error, but we can direct it elsewhere by assigning a pathname to the LD_DEBUG_OUTPUT environment variable.

If desired, we can assign multiple options to LD_DEBUG by separating them with commas (no spaces should appear). The output of the *symbols* option (which traces symbol resolution by the dynamic linker) is particularly voluminous.

LD_DEBUG is effective both for libraries implicitly loaded by the dynamic linker and for libraries dynamically loaded by *dlopen()*.

For security reasons, LD_DEBUG is (since *glibc* 2.2.5) ignored in set-user-ID and set-group-ID programs.

42.7 Summary

The dynamic linker provides the *dlopen* API, which allows programs to explicitly load additional shared libraries at run time. This allows programs to implement plug-in functionality.

An important aspect of shared library design is controlling symbol visibility, so that the library exports only those symbols (functions and variables) that should actually be used by programs linked against the library. We looked at a range of techniques that can be used to control symbol visibility. Among these techniques was the use of version scripts, which provide fine-grained control of symbol visibility.

We also showed how version scripts can be used to implement a scheme that allows a single shared library to export multiple definitions of a symbol for use by different applications linked against the library. (Each application uses the definition

that was current when the application was statically linked against the library.) This technique provides an alternative to the traditional library versioning approach of using major and minor version numbers in the shared library real name.

Defining initialization and finalization functions within a shared library allows us to automatically execute code when the library is loaded and unloaded.

The LD_PRELOAD environment variable allows us to preload shared libraries. Using this mechanism, we can selectively override functions and other symbols that the dynamic linker would normally find in other shared libraries.

We can assign various values to the LD_DEBUG environment variable in order to monitor the operation of the dynamic linker.

Further information

Refer to the sources of further information listed in Section 41.14.

42.8 Exercises

42-1. Write a program to verify that if a library is closed with *dlclose()*, it is not unloaded if any of its symbols are used by another library.

42-2. Add a *dladdr()* call to the program in Listing 42-1 (`dynload.c`) in order to retrieve information about the address returned by *dlsym()*. Print out the values of the fields of the returned *Dl_info* structure, and verify that they are as expected.