Rocky Enterprise Linux 9.2 Manual Pages on command 'dlopen.3'

$ man dlopen.3

DLOPEN(3) Linux Programmer's Manual DLOPEN(3)

NAME

dlclose, dlopen, dlmopen - open and close a shared object

SYNOPSIS

#include <dlfcn.h>

void *dlopen(const char *filename, int flags);

int dlclose(void *handle);

#define _GNU_SOURCE

#include <dlfcn.h>

void *dlmopen(Lmid_t lmid, const char *filename, int flags);

Link with -ldl.

DESCRIPTION

dlopen()

The function dlopen() loads the dynamic shared object (shared library) file named by the null-terminated string filename and returns an opaque "handle" for the loaded object. This handle is employed with other functions in the dlopen API, such as dlsym(3), dladdr(3), dlinfo(3), and dlclose().

If filename is NULL, then the returned handle is for the main program.

If filename contains a slash ("/"), then it is interpreted as a (rela?
tive or absolute) pathname. Otherwise, the dynamic linker searches for
the object as follows (see ld.so(8) for further details):

- (ELF only) If the calling object (i.e., the shared library or exe?
  cutable from which dlopen() is called) contains a DT_RPATH tag, and
  does not contain a DT_RUNPATH tag, then the directories listed in
  the DT_RPATH tag are searched.
- If, at the time that the program was started, the environment vari?
  able LD_LIBRARY_PATH was defined to contain a colon-separated list
  of directories, then these are searched. (As a security measure,
  this variable is ignored for set-user-ID and set-group-ID pro?
  grams.)
- (ELF only) If the calling object contains a DT_RUNPATH tag, then
  the directories listed in that tag are searched.
- The cache file /etc/ld.so.cache (maintained by ldconfig(8)) is
  checked to see whether it contains an entry for filename.
- The directories /lib and /usr/lib are searched (in that order).

If the object specified by filename has dependencies on other shared
objects, then these are also automatically loaded by the dynamic linker
using the same rules. (This process may occur recursively, if those
objects in turn have dependencies, and so on.)

One of the following two values must be included in flags:

RTLD_LAZY

Perform lazy binding. Resolve symbols only as the code that
references them is executed. If the symbol is never referenced,
then it is never resolved. (Lazy binding is performed only for
function references; references to variables are always immedi?
ately bound when the shared object is loaded.) Since glibc
2.1.1, this flag is overridden by the effect of the LD_BIND_NOW
environment variable.

RTLD_NOW

If this value is specified, or the environment variable

LD_BIND_NOW is set to a nonempty string, all undefined symbols in the shared object are resolved before dlopen() returns. If this cannot be done, an error is returned.

Zero or more of the following values may also be ORed in flags:

RTLD_GLOBAL

The symbols defined by this shared object will be made available for symbol resolution of subsequently loaded shared objects.

RTLD_LOCAL

This is the converse of RTLD_GLOBAL, and the default if neither flag is specified. Symbols defined in this shared object are not made available to resolve references in subsequently loaded shared objects.

RTLD_NODELETE (since glibc 2.2)

Do not unload the shared object during dlclose(). Consequently, the object's static and global variables are not reinitialized if the object is reloaded with dlopen() at a later time.

RTLD_NOLOAD (since glibc 2.2)

Don't load the shared object. This can be used to test if the object is already resident (dlopen() returns NULL if it is not, or the object's handle if it is resident). This flag can also be used to promote the flags on a shared object that is already loaded. For example, a shared object that was previously loaded with RTLD_LOCAL can be reopened with RTLD_NOLOAD | RTLD_GLOBAL.

RTLD_DEEPBIND (since glibc 2.3.4)

Place the lookup scope of the symbols in this shared object ahead of the global scope. This means that a self-contained ob? ject will use its own symbols in preference to global symbols with the same name contained in objects that have already been loaded.

If filename is NULL, then the returned handle is for the main program. When given to dlsym(3), this handle causes a search for a symbol in the main program, followed by all shared objects loaded at program startup, and then all shared objects loaded by dlopen() with the flag

RTLD_GLOBAL.

Symbol references in the shared object are resolved using (in order): symbols in the link map of objects loaded for the main program and its dependencies; symbols in shared objects (and their dependencies) that were previously opened with dlopen() using the RTLD_GLOBAL flag; and definitions in the shared object itself (and any dependencies that were loaded for that object).

Any global symbols in the executable that were placed into its dynamic symbol table by ld(1) can also be used to resolve references in a dy? namically loaded shared object. Symbols may be placed in the dynamic symbol table either because the executable was linked with the flag "-rdynamic" (or, synonymously, "--export-dynamic"), which causes all of the executable's global symbols to be placed in the dynamic symbol ta? ble, or because ld(1) noted a dependency on a symbol in another object during static linking.

If the same shared object is opened again with dlopen(), the same ob? ject handle is returned. The dynamic linker maintains reference counts for object handles, so a dynamically loaded shared object is not de? allocated until dlclose() has been called on it as many times as dlopen() has succeeded on it. Constructors (see below) are called only when the object is actually loaded into memory (i.e., when the refer? ence count increases to 1).

A subsequent dlopen() call that loads the same shared object with RTLD_NOW may force symbol resolution for a shared object earlier loaded with RTLD_LAZY. Similarly, an object that was previously opened with RTLD_LOCAL can be promoted to RTLD_GLOBAL in a subsequent dlopen().

If dlopen() fails for any reason, it returns NULL.

dlmopen()

This function performs the same task as dlopen()?the filename and flags arguments, as well as the return value, are the same, except for the differences noted below.

The dlmopen() function differs from dlopen() primarily in that it ac? cepts an additional argument, lmid, that specifies the link-map list

(also referred to as a namespace) in which the shared object should be loaded. (By comparison, dlopen() adds the dynamically loaded shared object to the same namespace as the shared object from which the dlopen() call is made.) The Lmid_t type is an opaque handle that refers to a namespace.

The lmid argument is either the ID of an existing namespace (which can be obtained using the dlinfo(3) RTLD_DI_LMID request) or one of the following special values:

LM_ID_BASE

Load the shared object in the initial namespace (i.e., the ap? plication's namespace).

LM_ID_NEWLM

Create a new namespace and load the shared object in that name? space. The object must have been correctly linked to reference all of the other shared objects that it requires, since the new namespace is initially empty.

If filename is NULL, then the only permitted value for lmid is LM_ID_BASE.

dlclose()

The function dlclose() decrements the reference count on the dynami? cally loaded shared object referred to by handle.

If the object's reference count drops to zero and no symbols in this object are required by other objects, then the object is unloaded after first calling any destructors defined for the object. (Symbols in this object might be required in another object because this object was opened with the RTLD_GLOBAL flag and one of its symbols satisfied a re? location in another object.)

All shared objects that were automatically loaded when dlopen() was in? voked on the object referred to by handle are recursively closed in the same manner.

A successful return from dlclose() does not guarantee that the symbols associated with handle are removed from the caller's address space. In addition to references resulting from explicit dlopen() calls, a shared

object may have been implicitly loaded (and reference counted) because of dependencies in other shared objects. Only when all references have been released can the shared object be removed from the address space.

RETURN VALUE

On success, dlopen() and dlmopen() return a non-NULL handle for the loaded object. On error (file could not be found, was not readable, had the wrong format, or caused errors during loading), these functions return NULL.

On success, dlclose() returns 0; on error, it returns a nonzero value.

Errors from these functions can be diagnosed using dlerror(3).

VERSIONS

dlopen() and dlclose() are present in glibc 2.0 and later. dlmopen() first appeared in glibc 2.3.4.

ATTRIBUTES

For an explanation of the terms used in this section, see at? tributes(7).

??

?Interface ? Attribute ? Value ?

??

?dlopen(), dlmopen(), dlclose() ? Thread safety ? MT-Safe ?

??

CONFORMING TO

POSIX.1-2001 describes dlclose() and dlopen(). The dlmopen() function is a GNU extension.

The RTLD_NOLOAD, RTLD_NODELETE, and RTLD_DEEPBIND flags are GNU exten? sions; the first two of these flags are also present on Solaris.

NOTES

dlmopen() and namespaces

A link-map list defines an isolated namespace for the resolution of symbols by the dynamic linker. Within a namespace, dependent shared objects are implicitly loaded according to the usual rules, and symbol references are likewise resolved according to the usual rules, but such resolution is confined to the definitions provided by the objects that

have been (explicitly and implicitly) loaded into the namespace.

The dlmopen() function permits object-load isolation?the ability to load a shared object in a new namespace without exposing the rest of the application to the symbols made available by the new object. Note that the use of the RTLD_LOCAL flag is not sufficient for this purpose, since it prevents a shared object's symbols from being available to any other shared object. In some cases, we may want to make the symbols provided by a dynamically loaded shared object available to (a subset of) other shared objects without exposing those symbols to the entire application. This can be achieved by using a separate namespace and the RTLD_GLOBAL flag.

The dlmopen() function also can be used to provide better isolation than the RTLD_LOCAL flag. In particular, shared objects loaded with RTLD_LOCAL may be promoted to RTLD_GLOBAL if they are dependencies of another shared object loaded with RTLD_GLOBAL. Thus, RTLD_LOCAL is in? sufficient to isolate a loaded shared object except in the (uncommon) case where one has explicit control over all shared object dependen? cies.

Possible uses of dlmopen() are plugins where the author of the plugin-loading framework can't trust the plugin authors and does not wish any undefined symbols from the plugin framework to be resolved to plugin symbols. Another use is to load the same object more than once. With? out the use of dlmopen(), this would require the creation of distinct copies of the shared object file. Using dlmopen(), this can be achieved by loading the same shared object file into different name? spaces.

The glibc implementation supports a maximum of 16 namespaces.

Initialization and finalization functions

Shared objects may export functions using the __attribute__((construc? tor)) and __attribute__((destructor)) function attributes. Constructor functions are executed before dlopen() returns, and destructor func? tions are executed before dlclose() returns. A shared object may ex? port multiple constructors and destructors, and priorities can be asso?

ciated with each function to determine the order in which they are exe?
cuted. See the gcc info pages (under "Function attributes") for fur?
ther information.

An older method of (partially) achieving the same result is via the use of two special symbols recognized by the linker: _init and _fini. If a dynamically loaded shared object exports a routine named _init(), then that code is executed after loading a shared object, before dlopen() returns. If the shared object exports a routine named _fini(), then that routine is called just before the object is unloaded. In this case, one must avoid linking against the system startup files, which contain default versions of these files; this can be done by using the gcc(1) -nostartfiles command-line option.

Use of _init and _fini is now deprecated in favor of the aforementioned constructors and destructors, which among other advantages, permit mul? tiple initialization and finalization functions to be defined.

Since glibc 2.2.3, atexit(3) can be used to register an exit handler that is automatically called when a shared object is unloaded.

History

These functions are part of the dlopen API, derived from SunOS.

BUGS

As at glibc 2.24, specifying the RTLD_GLOBAL flag when calling dl? mopen() generates an error. Furthermore, specifying RTLD_GLOBAL when calling dlopen() results in a program crash (SIGSEGV) if the call is made from any object loaded in a namespace other than the initial name? space.

EXAMPLES

The program below loads the (glibc) math library, looks up the address of the cos(3) function, and prints the cosine of 2.0. The following is an example of building and running the program:

```
$ cc dlopen_demo.c -ldl
$ ./a.out
-0.416147
```

Program source

```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
#include <gnu/lib-names.h>  /* Defines LIBM_SO (which will be a
                               string such as "libm.so.6") */
int
main(void)
{
    void *handle;
    double (*cosine)(double);
    char *error;
    handle = dlopen(LIBM_SO, RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(EXIT_FAILURE);
    }
    dlerror();    /* Clear any existing error */
    cosine = (double (*)(double)) dlsym(handle, "cos");
    /* According to the ISO C standard, casting between function
       pointers and 'void *', as done above, produces undefined results.
       POSIX.1-2001 and POSIX.1-2008 accepted this state of affairs and
       proposed the following workaround:
           *(void **) (&cosine) = dlsym(handle, "cos");
       This (clumsy) cast conforms with the ISO C standard and will
       avoid any compiler warnings.
       The 2013 Technical Corrigendum 1 to POSIX.1-2008 improved matters
       by requiring that conforming implementations support casting
       'void *' to a function pointer.  Nevertheless, some compilers
       (e.g., gcc with the '-pedantic' option) may complain about the
       cast used in this program. */
    error = dlerror();
    if (error != NULL) {
        fprintf(stderr, "%s\n", error);
```

```
            exit(EXIT_FAILURE);
        }
        printf("%f\n", (*cosine)(2.0));
        dlclose(handle);
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

ld(1), ldd(1), pldd(1), dl_iterate_phdr(3), dladdr(3), dlerror(3), dlinfo(3), dlsym(3), rtld-audit(7), ld.so(8), ldconfig(8)

gcc info pages, ld info pages

COLOPHON

This page is part of release 5.10 of the Linux man-pages project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man-pages/.