NAME

dladdr, dlclose, dlerror, dlopen, dlsym, dlvsym - programming interface to dynamic linking loader

SYNOPSIS

#include <dlfcn.h>

void *dlopen(const char *_filename_, int _flag_);

char *dlerror(void);

void *dlsym(void *_handle_, const char *_symbol_);

int dlclose(void *_handle_);

Link with *-ldl*.

DESCRIPTION

The four functions **dlopen**(), **dlsym**(), **dlclose**(), **dlerror**() implement the interface to the dynamic linking loader.

dlerror()

The function **dlerror**() returns a human-readable string describing the most recent error that occurred from **dlopen**(), **dlsym**() or **dlclose**() since the last call to **dlerror**(). It returns NULL if no errors have occurred since initialization or since it was last called.

dlopen()

The function **dlopen**() loads the dynamic library file named by the null-terminated string *filename* and returns an opaque handle for the dynamic library. If*filename* is NULL, then the returned handle is for the main program. If *filename* contains a slash (/), then it is interpreted as a (relative or absolute) pathname. Otherwise, the dynamic linker searches for the library as follows (see ld.so(8) for further details):

- (ELF only) If the executable file for the calling program 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 variable **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 programs.)
- (ELF only) If the executable file for the calling program 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 library has dependencies on other shared libraries, then these are also automatically loaded by the dynamic linker using the same rules. (This process may occur recursively, if those libraries in turn have dependencies, and so on.)

One of the following two values must be included in *flag*:

RTLD_LAZY

Perform lazy binding. Only resolve symbols 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 immediately bound when the library is loaded.)

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 library 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 *flag*:

RTLD_GLOBAL
The symbols defined by this library will be made available for symbol resolution of subsequently loaded libraries.

RTLD_LOCAL
This is the converse of **RTLD_GLOBAL**, and the default if neither flag is specified. Symbols defined in this library are not made available to resolve references in subsequently loaded libraries.

RTLD_NODELETE (since glibc 2.2)
Do not unload the library during **dlclose**(). Consequently, the library's static variables are not reinitialized if the library is reloaded with **dlopen**() at a later time. This flag is not specified in POSIX.1-2001.

RTLD_NOLOAD (since glibc 2.2)
Don't load the library. This can be used to test if the library is already resident (**dlopen**() returns NULL if it is not, or the library's handle if it is resident). This flag can also be used to promote the flags on a library that is already loaded. For example, a library that was previously loaded with **RTLD_LOCAL** can be reopened with **RTLD_NOLOAD | RTLD_GLOBAL**. This flag is not specified in POSIX.1-2001.

RTLD_DEEPBIND (since glibc 2.3.4)
Place the lookup scope of the symbols in this library ahead of the global scope. This means that a self-contained library will use its own symbols in preference to global symbols with the same name contained in libraries that have already been loaded. This flag is not specified in POSIX.1-2001.

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

External references in the library are resolved using the libraries in that library's dependency list and any other libraries previously opened with the **RTLD_GLOBAL** flag. If the executable was linked with the flag -rdynamic (or, synonymously, --export-dynamic), then the global symbols in the executable will also be used to resolve references in a dynamically loaded library.

If the same library is loaded again with **dlopen**(), the same library handle is returned. The dl library maintains reference counts for library handles, so a dynamic library is not deallocated until **dlclose**() has been called on it as many times as **dlopen**() has succeeded on it. The **_init**() routine, if present, is called only once. But a subsequent call with **RTLD_NOW** may force symbol resolution for a library earlier loaded with **RTLD_LAZY**.

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

dlsym()
The function **dlsym**() takes a handle of a dynamic library returned by **dlopen**() and the null-terminated symbol name, returning the address where that symbol is loaded into memory. If the symbol is not found, in the specified library or any of the libraries that were automatically loaded by **dlopen**() when that library was loaded, **dlsym**() returns NULL. (The search performed by **dlsym**() is breadth first through the dependency tree of these libraries.) Since the value of the symbol could actually be NULL (so that a NULL return from **dlsym**() need not indicate an error), the correct way to test for an error is to call **dlerror**() to clear any old error conditions, then call **dlsym**(), and then call **dlerror**() again, saving its return value into a variable, and check whether this saved value is not NULL.

There are two special pseudo-handles, **RTLD_DEFAULT** and **RTLD_NEXT**. The former will find the first occurrence of the desired symbol using the default library search order. The latter will find the next occurrence of a function in the search order after the current library. This

allows one to provide a wrapper around a function in another shared library.

dlclose()

The function **dlclose**() decrements the reference count on the dynamic library handle *handle*. If the reference count drops to zero and no other loaded libraries use symbols in it, then the dynamic library is unloaded.

The function **dlclose**() returns 0 on success, and nonzero on error.

The obsolete symbols _init() and _fini()

The linker recognizes special symbols **_init** and **_fini**. If a dynamic library exports a routine named **_init**(), then that code is executed after the loading, before **dlopen**() returns. If the dynamic library exports a routine named **_fini**(), then that routine is called just before the library is unloaded. In case you need to avoid linking against the system startup files, this can be done by using the **gcc(1)** *-nostartfiles* command-line option.

Using these routines, or the gcc **-nostartfiles** or **-nostdlib** options, is not recommended. Their use may result in undesired behavior, since the constructor/destructor routines will not be executed (unless special measures are taken).

Instead, libraries should export routines using the **__attribute__((constructor))** and **__attribute__((destructor))** function attributes. See the gcc info pages for information on these. Constructor routines are executed before**dlopen**() returns, and destructor routines are executed before **dlclose**() returns.

Glibc extensions: dladdr() and dlvsym()

Glibc adds two functions not described by POSIX, with prototypes

#define _GNU_SOURCE /* See feature_test_macros(7)
*/
#include <dlfcn.h>

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

void *dlvsym(void ****handle*, char ****symbol***, char ****version***);**

The function **dladdr**() takes a function pointer and tries to resolve name and file where it is located. Information is stored in the*Dl_info* structure:

```
typedef struct {
const char *dli_fname; /* Pathname of shared object that
contains address */
void *dli_fbase; /* Address at which shared object
is loaded */
const char *dli_sname; /* Name of symbol whose definition
overlaps addr */
void *dli_saddr; /* Exact address of symbol named
in dli_sname */
} Dl_info;
```

If no symbol matching *addr* could be found, then *dli_sname* and *dli_saddr* are set to NULL.

dladdr() returns 0 on error, and nonzero on success.

The function **dlvsym**(), provided by glibc since version 2.1, does the same as **dlsym**() but takes a version string as an additional argument.

CONFORMING TO

POSIX.1-2001 describes **dlclose**(), **dlerror**(), **dlopen**(), and **dlsym**().

NOTES

The symbols **RTLD_DEFAULT** and **RTLD_NEXT** are defined by <*dlfcn.h*> only when **_GNU_SOURCE** was defined before including it.

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

History

The dlopen interface standard comes from SunOS. That system also has **dladdr**(), but not **dlvsym**().

BUGS

Sometimes, the function pointers you pass to **dladdr**() may surprise you. On some architectures (notably i386 and x86_64), *dli_fname* and *dli_fbase* may end up pointing back at the object from which you called **dladdr**(), even if the function used as an argument should come from a dynamically linked library.

The problem is that the function pointer will still be resolved at compile time, but merely point to the *plt* (Procedure Linkage Table) section of the original object (which dispatches the call after asking the dynamic linker to resolve the symbol). To work around this, you can try to compile the code to be position-independent: then, the compiler cannot prepare the pointer at compile time anymore and today's **gcc(1)** will generate code that just loads the final symbol address from the *got* (Global Offset Table) at run time before passing it to **dladdr**().

EXAMPLE

Load the math library, and print the cosine of 2.0:

```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

int
main(int argc, char **argv)
{
void *handle;
double (*cosine)(double);
char *error;

handle = dlopen(libm.so, RTLD_LAZY);
if (!handle) {
fprintf(stderr, %sn, 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.1-2003 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 to POSIX.1-2008 (a.k.a.
POSIX.1-2013) 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, %sn, error);
exit(EXIT_FAILURE);
}

printf(%fn, (*cosine)(2.0));
dlclose(handle);
exit(EXIT_SUCCESS);
}
```

If this program were in a file named foo.c, you would build the program with the following command:

gcc -rdynamic -o foo foo.c -ldl

Libraries exporting **_init**() and **_fini**() will want to be compiled as follows, using *bar.c* as the example name:

gcc -shared -nostartfiles -o bar bar.c

SEE ALSO

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

ld.so info pages, gcc info pages, ld info pages

COLOPHON

This page is part of release 3.74 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 http://www.kernel.org/doc/man-pages/.