_dlfcn.h>

## NAME

dlfcn.h – dynamic linking

## SYNOPSIS

**#include <dlfcn.h>**

## DESCRIPTION

The <*dlfcn.h*> header shall define at least the following macros for use in the construction of a *dlopen*() *mode* argument:

RTLD_LAZY
: Relocations are performed at an implementation-defined time.

RTLD_NOW
: Relocations are performed when the object is loaded.

RTLD_GLOBAL
: All symbols are available for relocation processing of other modules.

RTLD_LOCAL
: All symbols are not made available for relocation processing by other modules.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int   dlclose(void *);
char  *dlerror(void);
void  *dlopen(const char *, int);
void  *dlsym(void *restrict, const char *restrict);
```

*The following sections are informative.*

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-2001, *dlopen*(), *dlclose*(), *dlsym*(), *dlerror*()

## COPYRIGHT

DLCLOSE

## PROLOG

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

## NAME

dlclose – close a dlopen object

## SYNOPSIS

```
#include <dlfcn.h>

int dlclose(void *handle);
```

## DESCRIPTION

The *dlclose*() function shall inform the system that the object referenced by a *handle* returned from a previous *dlopen*() invocation is no longer needed by the application.

The use of *dlclose*() reflects a statement of intent on the part of the process, but does not create any requirement upon the implementation, such as removal of the code or symbols referenced by *handle*. Once an object has been closed using *dlclose*() an application should assume that its symbols are no longer available to *dlsym*(). All objects loaded automatically as a result of invoking *dlopen*() on the referenced object shall also be closed if this is the last reference to it.

Although a *dlclose*() operation is not required to remove structures from an address space, neither is an implementation prohibited from doing so. The only restriction on such a removal is that no object shall be removed to which references have been relocated, until or unless all such references are removed. For instance, an object that had been loaded with a *dlopen*() operation specifying the RTLD_GLOBAL flag might provide a target for dynamic relocations performed in the processing of other objects-in such environments, an application may assume that no relocation, once made, shall be undone or remade unless the object requiring the relocation has itself been removed.

## RETURN VALUE

If the referenced object was successfully closed, *dlclose*() shall return 0. If the object could not be closed, or if *handle* does not refer to an open object, *dlclose*() shall return a non-zero value. More detailed diagnostic information shall be available through *dlerror*().

## ERRORS

No errors are defined.

*The following sections are informative.*

## EXAMPLES

The following example illustrates use of *dlopen*() and *dlclose*():

```
...
/* Open a dynamic library and then close it ... */


#include <dlfcn.h>
void *mylib;
int eret;


mylib = dlopen("mylib.so", RTLD_LOCAL | RTLD_LAZY);
...
eret = dlclose(mylib);
...
```

## APPLICATION USAGE

A conforming application should employ a *handle* returned from a *dlopen*() invocation only within a given scope bracketed by the *dlopen*() and *dlclose*() operations. Implementations are free to use

reference counting or other techniques such that multiple calls to *dlopen*() referencing the same object may return the same object for *handle*. Implementations are also free to reuse a *handle*. For these reasons, the value of a *handle* must be treated as an opaque object by the application, used only in calls to *dlsym*() and *dlclose*().

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

*dlerror*(), *dlopen*(), *dlsym*(), the Base Definitions volume of IEEE Std 1003.1-2001, <*dlfcn.h*>

## COPYRIGHT

DLERROR

## PROLOG

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

## NAME

dlerror – get diagnostic information

## SYNOPSIS

```
#include <dlfcn.h>

char *dlerror(void);
```

## DESCRIPTION

The *dlerror*() function shall return a null-terminated character string (with no trailing <newline>) that describes the last error that occurred during dynamic linking processing. If no dynamic linking errors have occurred since the last invocation of *dlerror*(), *dlerror*() shall return NULL. Thus, invoking *dlerror*() a second time, immediately following a prior invocation, shall result in NULL being returned.

The *dlerror*() function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

## RETURN VALUE

If successful, *dlerror*() shall return a null-terminated character string; otherwise, NULL shall be returned.

## ERRORS

No errors are defined.

*The following sections are informative.*

## EXAMPLES

The following example prints out the last dynamic linking error:

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

char *errstr;

errstr = dlerror();
if (errstr != NULL)
printf ("A dynamic linking error occurred: (%s)\n", errstr);
...
```

## APPLICATION USAGE

The messages returned by *dlerror*() may reside in a static buffer that is overwritten on each call to *dlerror*(). Application code should not write to this buffer. Programs wishing to preserve an error message should make their own copies of that message. Depending on the application environment with respect to asynchronous execution events, such as signals or other asynchronous computation sharing the address space, conforming applications should use a critical section to retrieve the error pointer and buffer.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

*dlclose*(), *dlopen*(), *dlsym*(), the Base Definitions volume of IEEE Std 1003.1-2001, <*dlfcn.h*>

## COPYRIGHT

DLOPEN

## PROLOG

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

## NAME

dlopen − gain access to an executable object file

## SYNOPSIS

```
#include <dlfcn.h>

void *dlopen(const char *file, int mode);
```

## DESCRIPTION

The *dlopen*() function shall make an executable object file specified by *file* available to the calling program. The class of files eligible for this operation and the manner of their construction are implementation-defined, though typically such files are executable objects such as shared libraries, relocatable files, or programs. Note that some implementations permit the construction of dependencies between such objects that are embedded within files. In such cases, a *dlopen*() operation shall load such dependencies in addition to the object referenced by *file*. Implementations may also impose specific constraints on the construction of programs that can employ *dlopen*() and its related services.

A successful *dlopen*() shall return a *handle* which the caller may use on subsequent calls to *dlsym*() and *dlclose*(). The value of this *handle* should not be interpreted in any way by the caller.

The *file* argument is used to construct a pathname to the object file. If *file* contains a slash character, the *file* argument is used as the pathname for the file. Otherwise, *file* is used in an implementation-defined manner to yield a pathname.

If the value of *file* is 0, *dlopen*() shall provide a *handle* on a global symbol object. This object shall provide access to the symbols from an ordered set of objects consisting of the original program image file, together with any objects loaded at program start-up as specified by that process image file (for example, shared libraries), and the set of objects loaded using a *dlopen*() operation together with the RTLD_GLOBAL flag. As the latter set of objects can change during execution, the set identified by *handle* can also change dynamically.

Only a single copy of an object file is brought into the address space, even if *dlopen*() is invoked multiple times in reference to the file, and even if different pathnames are used to reference the file.

The *mode* parameter describes how *dlopen*() shall operate upon *file* with respect to the processing of relocations and the scope of visibility of the symbols provided within *file*. When an object is brought into the address space of a process, it may contain references to symbols whose addresses are not known until the object is loaded. These references shall be relocated before the symbols can be accessed. The *mode* parameter governs when these relocations take place and may have the following values:

RTLD_LAZY
: Relocations shall be performed at an implementation-defined time, ranging from the time of the *dlopen*() call until the first reference to a given symbol occurs. Specifying RTLD_LAZY should improve performance on implementations supporting dynamic symbol binding as a process may not reference all of the functions in any given object. And, for systems supporting dynamic symbol resolution for normal process execution, this behavior mimics the normal handling of process execution.

RTLD_NOW
: All necessary relocations shall be performed when the object is first loaded. This may waste some processing if relocations are performed for functions that are never referenced. This behavior may be useful for applications that need to know as soon as an object is loaded that all symbols referenced during execution are available.

Any object loaded by *dlopen*() that requires relocations against global symbols can reference the symbols in the original process image file, any objects loaded at program start-up, from the object itself as

well as any other object included in the same *dlopen*() invocation, and any objects that were loaded in any *dlopen*() invocation and which specified the RTLD_GLOBAL flag. To determine the scope of visibility for the symbols loaded with a *dlopen*() invocation, the *mode* parameter should be a bitwise-inclusive OR with one of the following values:

RTLD_GLOBAL
: The object's symbols shall be made available for the relocation processing of any other object. In addition, symbol lookup using *dlopen*(*0, mode*) and an associated *dlsym*() allows objects loaded with this *mode* to be searched.

RTLD_LOCAL
: The object's symbols shall not be made available for the relocation processing of any other object.

If neither RTLD_GLOBAL nor RTLD_LOCAL are specified, then an implementation-defined default behavior shall be applied.

If a *file* is specified in multiple *dlopen*() invocations, *mode* is interpreted at each invocation. Note, however, that once RTLD_NOW has been specified all relocations shall have been completed rendering further RTLD_NOW operations redundant and any further RTLD_LAZY operations irrelevant. Similarly, note that once RTLD_GLOBAL has been specified the object shall maintain the RTLD_GLOBAL status regardless of any previous or future specification of RTLD_LOCAL, as long as the object remains in the address space (see *dlclose*()).

Symbols introduced into a program through calls to *dlopen*() may be used in relocation activities. Symbols so introduced may duplicate symbols already defined by the program or previous *dlopen*() operations. To resolve the ambiguities such a situation might present, the resolution of a symbol reference to symbol definition is based on a symbol resolution order. Two such resolution orders are defined: *load* or *dependency* ordering. Load order establishes an ordering among symbol definitions, such that the definition first loaded (including definitions from the image file and any dependent objects loaded with it) has priority over objects added later (via *dlopen*()). Load ordering is used in relocation processing. Dependency ordering uses a breadth-first order starting with a given object, then all of its dependencies, then any dependents of those, iterating until all dependencies are satisfied. With the exception of the global symbol object obtained via a *dlopen*() operation on a *file* of 0, dependency ordering is used by the *dlsym*() function. Load ordering is used in *dlsym*() operations upon the global symbol object.

When an object is first made accessible via *dlopen*() it and its dependent objects are added in dependency order. Once all the objects are added, relocations are performed using load order. Note that if an object or its dependencies had been previously loaded, the load and dependency orders may yield different resolutions.

The symbols introduced by *dlopen*() operations and available through *dlsym*() are at a minimum those which are exported as symbols of global scope by the object. Typically such symbols shall be those that were specified in (for example) C source code as having *extern* linkage. The precise manner in which an implementation constructs the set of exported symbols for a *dlopen*() object is specified by that implementation.

## RETURN VALUE

If *file* cannot be found, cannot be opened for reading, is not of an appropriate object format for processing by *dlopen*(), or if an error occurs during the process of loading *file* or relocating its symbolic references, *dlopen*() shall return NULL. More detailed diagnostic information shall be available through *dlerror*().

## ERRORS

No errors are defined.

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

*dlclose*(), *dlerror*(), *dlsym*(), the Base Definitions volume of IEEE Std 1003.1-2001, <*dlfcn.h*>

## COPYRIGHT

DLSYM

## PROLOG

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

## NAME

dlsym – obtain the address of a symbol from a dlopen object

## SYNOPSIS

```
#include <dlfcn.h>

void *dlsym(void *restrict handle, const char *restrict name);
```

## DESCRIPTION

The *dlsym*() function shall obtain the address of a symbol defined within an object made accessible through a *dlopen*() call. The *handle* argument is the value returned from a call to *dlopen*() (and which has not since been released via a call to *dlclose*()), and *name* is the symbol's name as a character string.

The *dlsym*() function shall search for the named symbol in all objects loaded automatically as a result of loading the object referenced by *handle* (see *dlopen*()). Load ordering is used in *dlsym*() operations upon the global symbol object. The symbol resolution algorithm used shall be dependency order as described in *dlopen*().

The RTLD_DEFAULT and RTLD_NEXT flags are reserved for future use.

## RETURN VALUE

If *handle* does not refer to a valid object opened by *dlopen*(), or if the named symbol cannot be found within any of the objects associated with *handle*, *dlsym*() shall return NULL. More detailed diagnostic information shall be available through *dlerror*() .

## ERRORS

No errors are defined.

*The following sections are informative.*

## EXAMPLES

The following example shows how *dlopen*() and *dlsym*() can be used to access either function or data objects. For simplicity, error checking has been omitted.

```
void    *handle;
int     *iptr, (*fptr)(int);


/* open the needed object */
handle = dlopen("/usr/home/me/libfoo.so", RTLD_LOCAL | RTLD_LAZY);


/* find the address of function and data objects */
*(void **)(&fptr) = dlsym(handle, "my_function");
iptr = (int *)dlsym(handle, "my_object");


/* invoke function, passing value of integer as a parameter */
(*fptr)(*iptr);
```

## APPLICATION USAGE

Special purpose values for *handle* are reserved for future use. These values and their meanings are:

RTLD_DEFAULT

The symbol lookup happens in the normal global scope; that is, a search for a symbol using this handle would find the same definition as a direct use of this symbol in the program code.

RTLD_NEXT

Specifies the next object after this one that defines *name*. *This one* refers to the object containing the invocation of *dlsym*(). The *next* object is the one found upon the application of a load order symbol resolution algorithm (see *dlopen*()). The next object is either one of global scope (because it was introduced as part of the original process image or because it was added with a *dlopen*() operation including the RTLD_GLOBAL flag), or is an object that was included in the same *dlopen*() operation that loaded this one.

The RTLD_NEXT flag is useful to navigate an intentionally created hierarchy of multiply-defined symbols created through *interposition*. For example, if a program wished to create an implementation of *malloc*() that embedded some statistics gathering about memory allocations, such an implementation could use the real *malloc*() definition to perform the memory allocation-and itself only embed the necessary logic to implement the statistics gathering function.

## RATIONALE

The ISO C standard does not require that pointers to functions can be cast back and forth to pointers to data. Indeed, the ISO C standard does not require that an object of type **void *** can hold a pointer to a function. Implementations supporting the XSI extension, however, do require that an object of type **void *** can hold a pointer to a function. The result of converting a pointer to a function into a pointer to another data type (except **void ***) is still undefined, however. Note that compilers conforming to the ISO C standard are required to generate a warning if a conversion from a **void *** pointer to a function pointer is attempted as in:

```
fptr = (int (*)(int))dlsym(handle, "my_function");
```

Due to the problem noted here, a future version may either add a new function to return function pointers, or the current interface may be deprecated in favor of two new functions: one that returns data pointers and the other that returns function pointers.

## FUTURE DIRECTIONS

None.

## SEE ALSO

*dlclose*(), *dlerror*(), *dlopen*(), the Base Definitions volume of IEEE Std 1003.1-2001, <*dlfcn.h*>

## COPYRIGHT