# Runtime Linker 3

## Overview

As part of the initialization and execution of a *dynamic executable*, an *interpreter* is called to complete the binding of the application to its shared object dependencies. In Solaris this interpreter is referred to as the runtime linker.

During the link-editing of a dynamic executable, a special *.interp* section, together with an associated program header, are created. This section contains a pathname specifying the program's interpreter. The default name supplied by the link-editor is that of the runtime linker - `/usr/lib/ld.so.1`.

During the process of executing a dynamic executable (see `exec(2)`) the kernel maps the file (see `mmap(2)`), and using the program header information (see "Program Header" on page 189), locates the name of the required interpreter. The kernel maps this interpreter and transfers control to it, passing sufficient information to allow the interpreter to continue binding the application and then run it.

In addition to initializing an application the runtime linker provides services that allow the application to extend its address space by mapping additional shared objects and binding to symbols within them.

The following is a simple breakdown of the runtime linkers functionality, and introduces the topics covered in this chapter:

- It analyzes the executable's dynamic information section (*.dynamic*) and determines what shared object dependencies are required.

- It locates and maps in these dependencies, and analyzes their dynamic information sections to determine if any additional shared object dependencies are required.
- Once all shared object dependencies are located and mapped, the runtime linker performs any necessary relocations to bind these objects in preparation for process execution.
- It calls any initialization functions provided by the shared object dependencies.
- It passes control to the application.
- During the application's execution, the runtime linker can be called upon to perform any delayed function binding.
- The application can also call upon the runtime linker's services to acquire additional shared objects by `dlopen(3X)`, and bind to symbols within these objects with `dlsym(3X)`.

## *Locating Shared Object Dependencies*

Usually, during the link-edit of a dynamic executable, one or more shared objects are explicitly referenced. These shared objects are recorded as dependencies within the dynamic executable (see "Shared Object Processing" on page 13 for more information).

The runtime linker first locates this dependency information and uses it to locate and map the associated shared objects. These shared object dependencies are processed in the same order as they were referenced during the link-edit of the executable.

Once all the dynamic executable's dependencies are mapped, they too are inspected, in the order they are mapped, to locate any additional shared object dependencies. This process continues until all dependent shared objects are located and mapped. This technique results in a breadth-first ordering of all dependent shared objects.

## *Directories Searched by the Runtime Linker*

The runtime linker knows of only one standard place to look for shared object dependencies, `/usr/lib`. Any dependency specified as a simple filename will be prefixed with this default directory name and the resulting pathname will be used to locate the actual file.

The *actual* shared object dependencies of any dynamic executable or shared object can be displayed using `ldd(1)`. For example, the file `/usr/bin/cat` has the following dependencies:

```
$ ldd /usr/bin/cat
        libintl.so.1 =>  /usr/lib/libintl.so.1
        libw.so.1 =>     /usr/lib/libw.so.1
        libc.so.1 =>     /usr/lib/libc.so.1
        libdl.so.1 =>    /usr/lib/libdl.so.1
```

Here, the file `/usr/bin/cat` has a dependency, or *needs*, the files `libintl.so.1`, `libw.so.1`, `libc.so.1` and `libdl.so.1`.

The shared object dependencies actually recorded in a file can be inspected by using the `dump(1)` command to display the file's *.dynamic* section, and referencing any entries that have a `NEEDED` tag. For example:

```
$ dump -Lvp /usr/bin/cat

/usr/bin/cat:
[INDEX] Tag      Value
[1]     NEEDED   libintl.so.1
[2]     NEEDED   libw.so.1
[3]     NEEDED   libc.so.1
.........
```

Notice that the dependency `libdl.so.1`, displayed in the previous `ldd(1)` example, is not recorded in the file `/usr/bin/cat`. This is because `ldd(1)` shows the *total* dependencies of the specified file, and `libdl.so.1` is actually a dependency of `/usr/lib/libc.so.1`.

In the previous `dump(1)` example the dependencies are expressed as simple filenames - in other words there is no '/' in the name. The use of a *simple* filename requires the runtime linker to build the required pathname from a set of rules. Filenames that contain an embedded '/' will be used as-is.

The simple filename recording is the standard, most flexible mechanism of recording dependencies, and is provided by using the `-l` option of the link-editor (see "Linking with Additional Libraries" on page 14, and "Naming Conventions" on page 84 for additional information on this topic).

Frequently, shared objects are distributed in a directory other than `/usr/lib`. If a dynamic executable or shared object needs to locate dependencies in another directory, the runtime linker must explicitly be told to search this directory.

The recommended way to indicate additional search paths to the runtime linker is to record a *runpath* during the link-edit of the dynamic executable or shared object (see "Directories Searched by the Runtime Linker" on page 18 for details on recording this information).

Any runpath recording can be displayed using `dump(1)` and referring to the entry that has the `RPATH` tag. For example:

```
$ dump -Lvp prog

prog:
[INDEX] Tag      Value
[1]     NEEDED   libfoo.so.1
[2]     NEEDED   libc.so.1
[3]     RPATH    /home/me/lib:/home/you/lib
.........
```

Here, `prog` has a dependency on `libfoo.so.1` and requires the runtime linker to search directories `/home/me/lib` and `/home/you/lib` before it looks in the default location `/usr/lib`.

Another way to add to the runtime linker's search path is to set the environment variable `LD_LIBRARY_PATH`. This environment variable (which is analyzed once at process startup) can be set to a colon-separated list of directories, and these directories will be searched by the runtime linker *before*

any runpath specification or default directory. This environment variable is well suited for debugging purposes such as forcing an application to bind to a local shared object. For example:

```
$ LD_LIBRARY_PATH=. prog
```

Here the file `prog` from the previous example will be bound to `libfoo.so.1` found in the present working directory.

Although useful as a *temporary* mechanism of influencing the runtime linker's search path, the use of this environment variable is strongly discouraged in production software. Any dynamic executables that can reference this environment variable will have their search paths augmented, which can result in an overall degradation in performance. Also, as pointed out in "Using an Environment Variable" on page 17, and "Directories Searched by the Runtime Linker" on page 18, this environment variable affects the link-editor.

If a shared object dependency cannot be located, `ldd(1)` will indicate that the object cannot be found, and any attempt to execute the application will result in an appropriate error message from the runtime linker:

```
$ ldd prog
        libfoo.so.1 =>    (not found)
        libc.so.1 =>      /usr/lib/libc.so.1
        libdl.so.1 =>     /usr/lib/libdl.so.1
$ prog
ld.so.1: prog: fatal: libfoo.so.1: can't open file: errno=2
```

**Note –** Any runtime linker error that results from the failure of an underlying system call will result in the system error code value being displayed as part of the associated diagnostic message. This value can be interpreted more fully by referencing `/usr/include/sys/errno.h`.

## *Relocation Processing*

Once the runtime linker has located and mapped all the shared object dependencies required by an application, it then processes each object and performs any necessary relocations.

# 3

During the link-editing of an object, any relocation information supplied with the input relocatable objects is applied to the output file. However, when building a dynamic executable or shared object, many of the relocations cannot be completed at link-edit time because they require *logical addresses* that are known only when the objects are mapped into memory. In these cases the link-editor generates new relocation records as part of the output file image, and it is this information that the runtime linker must now process.

For a more detailed description of the many relocation types, see "Relocation Types (Processor Specific)" on page 170. However, for the purposes of this discussion it is convenient to categorize relocations into one of two types:

- Non-symbolic relocations.
- Symbolic relocations.

The relocation records for an object can be displayed by using `dump(1)`. For example:

```
$ dump -rvp libbar.so.1

libbar.so.1:

.rela.got:
Offset      Symndx                Type              Addend

0x10438     0                     R_SPARC_RELATIVE  0
0x1043c     foo                   R_SPARC_GLOB_DAT  0
```

Here, the file `libbar.so.1` contains two relocation records that indicate that the *global offset table* (the *.got* section) must be updated.

The first relocation is a simple *relative* relocation that can be seen from its relocation type and from the fact that the symbol index (`Symndx`) field is zero. This relocation needs to use the base address at which the object was mapped into memory to update the associated *.got* offset.

The second relocation requires the address of the symbol `foo`. To complete this relocation the runtime linker must locate this symbol from the dynamic executable or shared objects that have so far been mapped.

## Symbol Lookup

When the runtime linker looks up a symbol, it does so by searching in each object, starting with the dynamic executable, and progressing through each shared object in the same order in which the objects were mapped.

As discussed in previous sections, `ldd(1)` will list the shared object dependencies of a dynamic executable in the order in which they are mapped. Therefore, if the shared object `libbar.so.1` requires the address of symbol `foo` to complete its relocation, and this shared object is a dependency of the dynamic executable `prog`:

```
$ ldd prog
        libfoo.so.1 =>   /home/me/lib/libfoo.so.1
        libbar.so.1 =>   /home/me/lib/libbar.so.1
```

Then, the runtime linker will first look for `foo` in the dynamic executable `prog`, then in the shared object `/home/me/lib/libfoo.so.1`, and finally in the shared object `/home/me/lib/libbar.so.1`.

**Note –** Symbol lookup can be an expensive operation, especially as the size of symbol names increases, and the numbers of shared object dependencies increase. This aspect of performance is discussed in more detail in "Performance Considerations" on page 96.

## Interposition

The runtime linkers mechanism of searching for a symbol first in the dynamic executable and then in each of the shared object dependencies means that the *first* occurrence of the required symbol will satisfy the search. Therefore, if more than one instance of the same symbol exists, the first instance will *interpose* on all others.

## When Relocations are Performed

Having briefly described the relocation process, together with the simplification of relocations into the two types, non-symbolic and symbolic, it is also useful to distinguish relocations by when they are performed. This distinction arises due to the type of *reference* being made to the relocated offset, and can be either:

- A data reference.
- A function reference.

A *data reference* refers to an address that is used as a data item by the application code. The runtime linker has no knowledge of the application code, and so does not know when this data item will be referenced. Therefore, all data relocations must be carried out during process initialization, before the application gains control.

A *function reference* refers to the address of a function that will be called by the application code. During the compilation and link-editing of any dynamic module, calls to global functions are relocated to become calls to a *procedure linkage table* entry (these entries make up the *.plt* section).

These *.plt* entries are constructed so that when first called control is passed to the runtime linker. The runtime linker will look up the required symbol and rewrite information in the application so that any future calls to this *.plt* entry will go directly to the function.This mechanism allows relocations of this type to be *deferred* until the first instance of a function being called, a process that is sometimes referred to as *lazy* binding.

The runtime linker's default mode of performing lazy binding can be overridden by setting the environment variable `LD_BIND_NOW` to any non-null value. This environment variable setting causes the runtime linker to perform both data reference and function reference relocations during process initialization, before transferring control to the application. For example:

```
$ LD_BIND_NOW=yes prog
```

Here, *all* relocations within the file `prog` and within its shared object dependencies will be processed before control is transferred to the application.

## Relocation Errors

The most common relocation error occurs when a symbol cannot be found. This condition will result in an appropriate runtime linker error message and the termination of the application. For example:

```
$ ldd prog
        libfoo.so.1 =>    ./libfoo.so.1
        libc.so.1 =>      /usr/lib/libc.so.1
        libbar.so.1 =>    ./libbar.so.1
        libdl.so.1 =>     /usr/lib/libdl.so.1
$ prog
ld.so.1: prog: fatal: relocation error: symbol not found: bar: \
referenced in ./libfoo.so.1
```

Here the symbol `bar`, which is referenced in the file `libfoo.so.1`, can not be located.

---

**Note –** During the link-edit of a dynamic executable any potential relocation errors of this sort will be flagged as fatal undefined symbols (see "Generating an Executable" on page 28 for examples). This runtime relocation error can occur if the link-edit of `main` used a different version of the shared object `libbar.so.1` that contained a symbol definition for `bar`, or if the `-z nodefs` option was used as part of the link-edit.

---

If a relocation error of this type occurs because a symbol used as a data reference cannot be located, the error condition will occur immediately during process initialization. However, because of the default mode of lazy binding, if a symbol used as a function reference cannot be found, the error condition will occur after the application has gained control.

This latter case can take minutes or months, or might never occur, depending on the execution paths exercised throughout the code. To guard against errors of this kind, the relocation requirements of any dynamic executable or shared object can be validated using `ldd(1)`.

When the -d option is specified with ldd(1), all shared object dependencies will be printed and all data reference relocations will be processed. If a data reference cannot be resolved, a diagnostic message will be produced. From the previous example this reveals:

```
$ ldd -d prog
        libfoo.so.1 =>    ./libfoo.so.1
        libc.so.1 =>      /usr/lib/libc.so.1
        libbar.so.1 =>    ./libbar.so.1
        libdl.so.1 =>     /usr/lib/libdl.so.1
        symbol not found: bar           (./libfoo.so.1)
```

When the -r option is specified with ldd(1), all data *and* function reference relocations will be processed, and if either cannot be resolved a diagnostic message will be produced.

## *Loading Additional Objects*

The previous sections have described how the runtime linker initializes a process from the dynamic executable and its shared object dependencies as they were defined during the link-editing of each module. The runtime linker also provides an additional level of flexibility by allowing you to introduce new objects during process initialization.

The environment variable LD_PRELOAD can be initialized to a shared object or relocatable object filename, or a string of filenames separated by white space. These objects are mapped *after* the dynamic executable and *before* any shared object dependencies. For example:

```
$ LD_PRELOAD=./newstuff.so.1 prog
```

Here the dynamic executable `prog` will be mapped, followed by the shared object `newstuff.so.1`, and then by the shared object dependencies defined within `prog`. The order in which these objects are processed can be displayed using `ldd(1)`:

```
$ LD_PRELOAD=./newstuff.so.1 ldd prog
        ./newstuff.so.1 => ./newstuff.so
        libc.so.1 =>     /usr/lib/libc.so.1
```

Another example is:

```
$ LD_PRELOAD="./foo.o ./bar.o" prog
```

Here the preloading is a little more complex and time consuming. The runtime linker first link-edits the relocatable objects `foo.o` and `bar.o` to generate a shared object that is maintained in memory. This memory image is then inserted between the dynamic executable and the normal shared object dependencies in exactly the same manner as the shared object `newstuff.so.1` was preloaded in the previous example. Again, the order in which these objects are processed can be displayed with `ldd(1)`:

```
$ LD_PRELOAD="./foo.o ./bar.o" ldd prog
        ./foo.o =>       ./foo.o
        ./bar.o =>       ./bar.o
        libc.so.1 =>     /usr/lib/libc.so.1
```

These mechanisms of inserting a shared object after a dynamic executable take the concept of *interposition*, introduced on page 59, to another level. Using these mechanisms, it is possible to experiment with a new implementation of a function that resides in a standard shared object. By preloading just that function it will interpose on the original. Thus the old functionality can be completely hidden with the new preloaded version.

Another use of preloading is to augment a function that resides in a standard shared object. Here the intention is to have the new symbol interpose on the original, allowing the new function to carry out some additional processing, while still having it call through to the original function. This mechanism

requires either a symbol alias to be associated with the original function (see "Simple Resolutions" on page 22), or the ability to look up the original symbol's address (see "Using Interposition" on page 75).

## *Initialization and Termination Routines*

Before transferring control to the application, the runtime linker processes any initialization (*.init*) and termination (*.fini*) sections found in any of the shared object dependencies. These sections, and the symbols that describe them, were created during the link-editing of the shared objects (see "Initialization and Termination Sections" on page 19).

Any initialization routines for shared object dependencies are called in *reverse* load order - in other words, the reverse order of the shared objects displayed with `ldd(1)`.

Any termination routines for shared object dependencies are organized such that they can be recorded by `atexit(3C)`. Termination routines are therefore called in load order when the process calls `exit(2)`.

Although this initialization and termination calling sequence seems quite straightforward, be careful about placing too much emphasis on this sequence, as the ordering of shared objects can be affected by both shared object and application development (see "Dependency Ordering" on page 90 for more details).

**Note –** Any *.init* or *.fini* sections within the dynamic executable are called from the application itself by the process start-up and termination mechanism supplied by the compiler driver. The dynamic executable's *.init* section is called last, after all the shared object dependency's *.init* sections are executed. The dynamic executable's *.fini* section is called first, before the shared object dependency's *.fini* sections are executed.

## *Security*

Secure processes have some restrictions applied to the evaluation of their dependencies to prevent malicious dependency substitution or symbol interposition.

The runtime linker categorizes a process as secure if the user is not the *root*, and either the real users and effective users identifiers are not equal (see `getuid(2)` and `geteuid(2)`), or the real group and effective group identifiers are not equal (see `getgid(2)` and `getegid(2)`).

If an `LD_LIBRARY_PATH` environment variable is in effect (see "Directories Searched by the Runtime Linker" on page 55) for a secure process, then only the *trusted* directories specified by this variable will be used to augment the runtime linker's search rules. Presently, the only trusted directory known to the runtime linker is `/usr/lib`.

In a secure process, any *runpath* specifications provided by the application or any of it's dependencies (see "Directories Searched by the Runtime Linker" on page 55), will be used provided they are full pathnames - in other words the pathname starts with a '/'.

Additional objects may be loaded with a secure process using the `LD_PRELOAD` environment variable (see "Loading Additional Objects" on page 67) provided the objects are specified as simple filenames - in other words there is no '/' in the name. These objects will be located subject to the search path restrictions previously described.

In a secure process, any dependencies that consist of simple filenames will be processed using the pathname restrictions outlined above. Dependencies that are expressed as full or relative pathnames will be used as is. Therefore, the developer of a secure process should insure that the target directory referenced as a full or relative pathname dependency is suitably protected from malicious intrusion.

When creating a secure process, it is recommended that relative pathnames *not* be used to express dependencies or to construct `dlopen(3x)` pathnames. This restriction should be applied to the application and to *all* dependencies.

## *Runtime Linking Programming Interface*

The previous discussions described how the shared object dependencies specified during the link-edit of an application are processed by the runtime linker during process initialization. In addition to this mechanism, the application can extend its address space during its execution by binding to additional shared objects. This extensibility is provided by allowing the

application to request the same services of the runtime linker as used to process the shared object's dependencies specified during the link-edit of the application.

This delayed object binding has several advantages:

- By processing a shared object when it is required rather than during the initialization of an application, start-up time can be greatly reduced. In fact, the shared object might not be required if its services are not needed during a particular run of the application, such as for help or debugging information.
- The application can choose between several different shared objects depending on the exact services required, such as for a networking protocol.
- Any shared objects added to the process address space during execution can be freed after use.

The following is a typical scenario that an application can perform to access an additional shared object, and introduces the topics covered in the next sections:

- A shared object is located and added to the address space of a running application using `dlopen(3X)`. Any dependencies this shared object has are located and added at this time.
- The shared object(s) added are relocated, and any initialization sections within the new shared object(s) are called.
- The application locates symbols within the added shared object(s) using `dlsym(3X)`. The application can then reference the data or call the functions defined by these new symbols.
- After the application has finished with the shared object(s) the address space can be freed using `dlclose(3X)`. Any termination sections within the shared object(s) being freed will be called at this time.
- Any error conditions that occur as a result of using these runtime linker interface routines can be displayed using `dlerror(3X)`.

The services of the runtime linker are defined in the header file `dlfcn.h` and are made available to an application by the shared object `libdl.so.1`. For example:

```
$ cc -o prog main.c -ldl
```

Here the file `main.c` can make reference to any of the `dlopen(3X)` family of routines, and the application `prog` will be bound to these routines at runtime.

## *Loading Additional Objects*

Additional shared objects can be added to a running process's address space using `dlopen(3X)`. This function takes a *filename* and a *binding mode* as arguments, and returns a *handle* to the application. This handle can be used to locate symbols for use by the application using `dlsym(3X)`.

If the filename is specified as a simple filename - in other words, there is no '/' in the name, then the runtime linker will use a set of rules to build an appropriate pathname. Filenames that contain a '/' will be used as-is.

These search path rules are exactly the same as are used to locate any initial shared object dependencies (see "Directories Searched by the Runtime Linker" on page 55). For example, if the file `main.c` contains the following code fragment:

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

main(int argc, char ** argv)
{
    void *  handle;
    .....

    if ((handle = dlopen("foo.so.1", RTLD_LAZY)) == NULL) {
            (void) printf("dlopen: %s\n", dlerror());
            exit (1);
    }
    .....
```

then to locate the shared object `foo.so.1`, the runtime linker will use any `LD_LIBRARY_PATH` definition present at process initialization, followed by any runpath specified during the link-edit of `prog`, and finally the default location `/usr/lib`.

If the filename is specified as:

```
if ((handle = dlopen("./foo.so.1", RTLD_LAZY)) == NULL) {
```

then the runtime linker will search for the file only in the present working directory.

**Note –** It is recommended that any shared object specified using `dlopen(3X)` be referenced by its *versioned* filename (for more information on versioning see "Coordination of Versioned Filenames" on page 136).

If the required shared object cannot be located, `dlopen(3X)` will return a `NULL` handle. In this case `dlerror(3X)` can be used to display the true reason for the failure. For example:

```
$ cc -o prog main.c -ldl
$ prog
dlopen: ld.so.1: prog: fatal: foo.so.1: can't open file: errno=2
```

The `errno` value can be referenced in `/usr/include/sys/errno.h`.

If the shared object being added by `dlopen(3X)` has dependencies on other shared objects, they too will be brought into the process's address space.

If the shared object specified by `dlopen(3X)`, or any of its dependencies, are already part of the process image, then the shared objects will not be processed any further, but a valid handle will still be returned to the application. This mechanism prevents the same shared object from being mapped more than once, and allows an application to obtain a handle to itself. For example, if the `main.c` example contained the following code:

```
if ((handle = dlopen((const char *)0, RTLD_LAZY)) == NULL) {
```

then the handle returned from `dlopen(3X)` can be used to locate symbols within the application itself, within any of the shared object dependencies loaded as part of the process's initialization, or within any objects added to the process's address space using a `dlopen(3X)` that specified the `RTLD_GLOBAL` flag.

## Relocation Processing

As described in "Relocation Processing" on page 57, after locating and mapping any shared objects, the runtime linker must process each object and perform any necessary relocations. Any shared objects brought into the process's address space with `dlopen(3X)` must also be relocated in the same manner.

For simple applications this process might be quite uninteresting. However, for users who have more complex applications with many `dlopen(3X)` calls involving many shared objects, possibly with common dependencies, this topic can be quite important.

Relocations can be categorized according to when they occur. The default behavior of the runtime linker is to process all data reference relocations at initialization and all function references during process execution, a mechanism commonly referred to as *lazy* binding.

This same mechanism is applied to any shared objects added with `dlopen(3X)` when the *mode* is defined as `RTLD_LAZY`. An alternative is to require all relocations of a shared object to be performed immediately when the shared object is added, and this can be achieved by using a *mode* of `RTLD_NOW`.

Relocations can also be categorized into non-symbolic and symbolic. The remainder of this section covers issues regarding symbolic relocations, regardless of when these relocations occur, with a focus on some of the subtleties of symbol lookup.

### Symbol Lookup

If a shared object acquired by `dlopen(3X)` refers to a global symbol, the runtime linker will locate this symbol in the same manner as any other symbol lookup.

The runtime linker will first look in the dynamic executable, and then look in each of the shared objects provided during the initialization of the process. However, if the symbol is still not found, the runtime linker will continue the search, looking in the shared object acquired through the `dlopen(3X)` and in any of its dependencies.

For example, let's take the dynamic executable `prog`, and the shared object `B.so.1`, each of which has the following (simplified) dependencies:

```
$ ldd prog
        A.so.1 =>        ./A.so.1
$ ldd B.so.1
        C.so.1 =>        ./C.so.1
```

If `prog` acquires the shared object `B.so.1` by `dlopen(3X)`, then any symbol required to relocate the shared objects `B.so.1` and `C.so.1` will first be looked for in `prog`, followed by `A.so.1`, followed by `B.so.1`, and finally in `C.so.1`.

In this simple case, it might be easier to think of the shared objects acquired through the `dlopen(3X)` as if they had been added to the end of the original link-edit of the application. For example, the objects referenced above can be expressed diagrammatically:

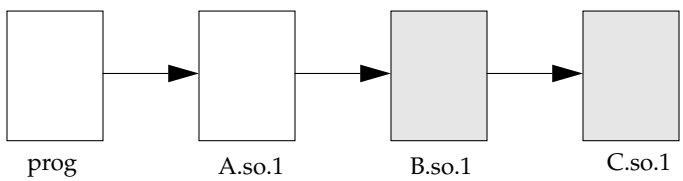


*Figure 3-1* A Single dlopen(3X) Request

Any symbol lookup required by the objects acquired from the `dlopen(3X)`, shown as shaded blocks, will proceed from the dynamic executable `prog` through to the final shared object `C.so.1`.

**Note –** Objects added to the process address space do not affect the normal symbol lookup required by either the application or its initial shared object dependencies. For example, if `A.so.1` requires a function relocation *after* the above `dlopen(3X)` has occurred, the runtime linker's normal search for the relocation symbol will be to look in `prog` and then `A.so.1`, but not to follow through and look in `B.so.1` or `C.so.1`.

This symbol lookup algorithm is established by assigning lookup *scopes* to each object. These scopes maintain associations between objects based on their introduction into the process address space, and on any dependency relationships between the objects.

All objects obtained during the process's initialization are assigned a *global* scope. Any object within the global scope can be used by any other object to provide symbols for relocation.

The shared objects associated with a given `dlopen(3X)` are assigned a unique *local* scope that insures that only objects associated with the same `dlopen(3X)` are allowed to look up symbols within themselves and their related dependencies.

This concept of defining associations between objects becomes more clear in applications that carry out more than one `dlopen(3X)`. For example, if the shared object `D.so.1` has the following dependency:

```
$ ldd D.so.1
        E.so.1 =>          ./E.so.1
```

and the `prog` application was to `dlopen(3X)` this shared object in addition to the shared object `B.so.1`, then diagrammatically the symbol lookup relationship between the objects can be represented as:

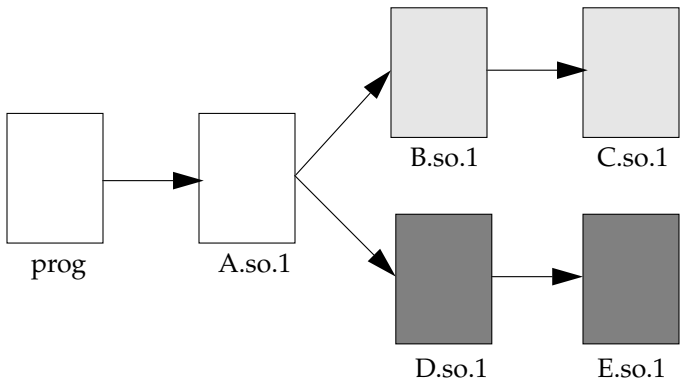


*Figure 3-2* Multiple dlopen(3X) Requests

If both `B.so.1` and `D.so.1` contain a definition for the symbol `foo`, and both `C.so.1` and `E.so.1` contain a relocation that requires this symbol, then because of the association of objects defined by the runtime linker, `C.so.1` will be bound to the definition in `B.so.1`, and `E.so.1` will be bound to the definition in `D.so.1`. This mechanism is intended to provide the most intuitive binding of shared objects obtained from multiple calls to `dlopen(3X)`.

When shared objects are used in the scenarios that have so far been described, the order in which each `dlopen(3X)` occurs has no effect on the resulting symbol binding. However, when shared objects have common dependencies the resultant bindings can be affected by the order in which the `dlopen(3X)` calls are made.

Take for example the shared objects `O.so.1` and `P.so.1`, which have the same common dependency:

```
$ ldd O.so.1
        Z.so.1 =>        ./Z.so.1
$ ldd P.so.1
        Z.so.1 =>        ./Z.so.1
```

In this example, the `prog` application will `dlopen(3X)` each of these shared objects. Because the shared object `Z.so.1` is a common dependency of both `O.so.1` and `P.so.1` it will be assigned both of the local scopes that are associated with the two `dlopen(3X)` calls. Diagrammatically this can be represented as:

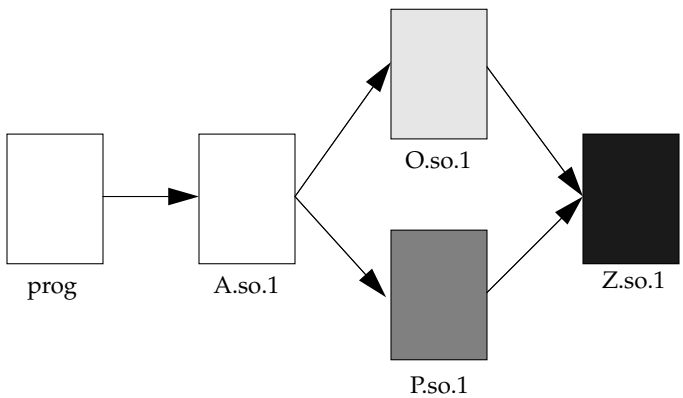


*Figure 3-3* Multiple dlopen(3X) Requests With A Common Dependency

The result is that `Z.so.1` will be available for both `O.so.1` and `P.so.1` to look up symbols, but more importantly, as far as `dlopen(3X)` ordering is concerned, `Z.so.1` will also be able to look up symbols in both `O.so.1` and `P.so.1`.

Therefore, if both `O.so.1` and `P.so.1` contain a definition for the symbol `foo` which is required for a `Z.so.1` relocation, the actual binding that occurs is unpredictable because it will be affected by the order of the `dlopen(3X)` calls. If the functionality of symbol `foo` differs between the two shared objects in which it is defined, the overall outcome of executing code within `Z.so.1` might vary depending on the application's `dlopen(3x)` ordering.

There is one final convolution involving the mode of a `dlopen(3X)`. All previous examples have revolved around the shared objects obtained by a `dlopen(3X)` each having a unique local scope, or a combination of local scopes if a shared object is a common dependency. It is also possible to give a shared object a *global* scope by augmenting the mode argument with the `RTLD_GLOBAL` flag. Under this mode, any shared objects obtained through a `dlopen(3X)` can be used by any other objects to locate symbols.

In addition, any object obtained by `dlopen(3X)` with the `RTLD_GLOBAL` flag will also be available for symbol lookup using `dlopen(0)` (see "Loading Additional Objects" on page 67).

## *Obtaining New Symbols*

A process can obtain the address of a specific symbol using `dlsym(3X)`. This function takes a *handle* and a *symbol name*, and returns the address of the symbol to the caller. The *handle* directs the search for the symbol in the following manner:

- The *handle* returned from a `dlopen(3X)` of a *named* shared object will allow symbols to be obtained from that shared object, or from any of its dependencies.
- The *handle* returned from a `dlopen(3X)` of a file whose value is 0 will allow symbols to be obtained from the dynamic executable, from any of its initialization dependencies, or from any object obtained by a `dlopen(3X)` with the `RTLD_GLOBAL` mode.
- The special *handle* `RTLD_NEXT` will allow symbols to be obtained from the *next* associated shared object.

The first example is probably the most common. Here an application will add additional shared objects to its address space and use `dlsym(3X)` to locate function or data symbols. The application then uses these symbols to call upon services provided in these new shared objects. For example, let's take the file `main.c` that contains the following code:

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

main()
{
    void *  handle;
    int *   dptr, (* fptr)();

    if ((handle = dlopen("foo.so.1", RTLD_LAZY)) == NULL) {
            (void) printf("dlopen: %s\n", dlerror());
            exit (1);
    }

    if (((fptr = (int (*)())dlsym(handle, "foo")) == NULL) ||
        ((dptr = (int *)dlsym(handle, "bar")) == NULL)) {
            (void) printf("dlsym: %s\n", dlerror());
            exit (1);
    }

    return ((*fptr)(*dptr));
}
```

Here the symbols `foo` and `bar` will be searched for in the file `foo.so.1` followed by any shared object dependencies that are associated with this file. The function `foo` is then called with the single argument `bar` as part of the `return` statement.

If the application `prog` is built using the above file `main.c`, and its initial shared object dependencies are:

```
$ ldd prog
        libdl.so.1 =>    /usr/lib/libdl.so.1
        libc.so.1 =>     /usr/lib/libc.so.1
```

then if the filename specified in the `dlopen(3X)` had the value 0, the symbols `foo` and `bar` will be searched for in `prog`, followed by `/usr/lib/libdl.so.1`, and finally `/usr/lib/libc.so.1`.

Once the *handle* has indicated the root at which to start a symbol search, the search mechanism follows the same model as was described in "Symbol Lookup" on page 59.

If the required symbol cannot be located, `dlsym(3X)` will return a `NULL` value. In this case `dlerror(3X)` can be used to indicate the true reason for the failure. For example;

```
$ prog
dlsym: ld.so.1: main: fatal: dlsym: can't find symbol bar
```

Here the application `prog` was unable to locate the symbol `bar`.

## *Using Interposition*

The special handle `RTLD_NEXT` allows an application to locate the next symbol in a symbol scope. For example, if the application `prog` contained the following code fragment:

```
if ((fptr = (int (*)())dlsym(RTLD_NEXT, "foo")) == NULL) {
        (void) printf("dlsym: %s\n", dlerror());
        exit (1);
}

return ((*fptr)());
```

then `foo` will be searched for in the shared objects associated with `prog`, in this case, `/usr/lib/libdl.so.1` and then `/usr/lib/libc.so.1`. If this code fragment was contained in the file `B.so.1` from the example shown in Figure 3-2 on page 71, then `foo` will be searched for in the associated shared object `C.so.1` only.

Using RTLD_NEXT provides a means to exploit symbol *interposition*. For example, a shared object function can be interposed upon by a preceding shared object, which can then augment the processing of the original function. If the following code fragment is placed in the shared object malloc.so.1:

```
#include    <sys/types.h>
#include    <dlfcn.h>
#include    <stdio.h>

void *
malloc(size_t size)
{
    static void * (* fptr)() = 0;
    char             buffer[50];

    if (fptr == 0) {
        fptr = (void * (*)())dlsym(RTLD_NEXT, "malloc");
        if (fptr == NULL) {
                (void) printf("dlopen: %s\n", dlerror());
                return (0);
        }
    }

    (void) sprintf(buffer, "malloc: %#x bytes\n", size);
    (void) write(1, buffer, strlen(buffer));
    return ((*fptr)(size));
}
```

Then by interposing this shared object between the system library /usr/lib/libc.so.1 where malloc(3C) usually resides, any calls to this function will be interposed on before the original function is called to complete the allocation:

```
$ cc -o malloc.so.1 -G -K pic malloc.c
$ cc -o prog file1.o file2.o ..... -R. malloc.so.1
$ prog
malloc: 0x32 bytes
malloc: 0x14 bytes
..........
```

Alternatively, this same interposition can be achieved by:

```
$ cc -o malloc.so.1 -G -K pic malloc.c
$ cc -o prog main.c
$ LD_PRELOAD=./malloc.so.1 prog
malloc: 0x32 bytes
malloc: 0x14 bytes
..........
```

**Note –** Users of any interposition technique must be careful to handle any possibility of recursion. The previous example formats the diagnostic message using `sprintf(3S)`, instead of using `printf(3S)` directly, to avoid any recursion caused by `printf(3S)`'s use of `malloc(3C)`.

The use of `RTLD_NEXT` within a dynamic executable or preloaded shared object provides a predictable and useful interpositioning technique. However, care should be taken when using this technique in a generic shared object dependency, as the actual load order of shared objects is not always predictable (see "Dependency Ordering" on page 90).

## *Debugging Aids*

Provided with the Solaris linkers is a debugging library that allows you to trace the runtime linking process in more detail. This library helps you understand, or debug, the execution of applications or libraries. This is a *visual* aid, and although the type of information displayed using this library is expected to remain constant, the exact format of the information might change slightly from release to release.

Some of the debugging output might be unfamiliar to those who do not have an intimate knowledge of the runtime linker. However, many aspects can be of general interest to you.

Debugging is enabled by using the environment variable `LD_DEBUG`. All debugging output is prefixed with the process identifier and by default is directed to the standard error. This environment variable must be augmented with one or more tokens to indicate the type of debugging required.

The tokens available with this debugging option can be displayed by using `LD_DEBUG=help`. Any dynamic executable can be used to solicit this information, as the process itself will terminate following the display of the information. For example:

```
$ LD_DEBUG=help prog
11693:
11693:          For debugging the run-time linking of an application:
11693:                 LD_DEBUG=option1,option2  prog
11693:          enables diagnostics to the stderr.  The additional
11693:          option:
11693:                 LD_DEBUG_OUTPUT=file
11693:          redirects the diagnostics to an output file created
11593:          using the specified name and the process id as a
11693:          suffix.  All output is prepended with the process id.
11693:
11693:
11693: bindings  display symbol binding; detail flag shows
11693:          absolute:relative addresses
11693: detail    provide more information in conjunction with other
11693:          options
11693: files     display input file processing (files and libraries)
11693: help      display this help message
11693: libs      display library search paths
11693: reloc     display relocation processing
11693: symbols   display symbol table processing;
11693:          detail flag shows resolution and linker table addition
11693: versions  display version processing
$
```

**Note –** The above is an example, and shows the options meaningful to the runtime linker. The exact options might differ from release to release.

The environment variable `LD_DEBUG_OUTPUT` can be used to specify an output file for use instead of the standard error. The output file name will be suffixed with the process identifier.

Debugging of secure applications is not allowed.

One of the most useful debugging options is to display the symbol bindings that occur at runtime. For example, let's take a very trivial dynamic executable that has a dependency on two local shared objects:

```
$ cat bar.c
int bar = 10;
$ cc -o bar.so.1 -Kpic -G bar.c

$ cat foo.c
foo(int data)
{
        return (data);
}
$ cc -o foo.so.1 -Kpic -G foo.c

$ cat main.c
extern  int     foo();
extern  int     bar;

main()
{
        return (foo(bar));
}
$ cc -o prog main.c -R/tmp:. foo.so.1 bar.so.1
```

The runtime symbol bindings can be displayed by setting `LD_DEBUG=bindings`:

```
$ LD_DEBUG=bindings prog
11753: .......
11753: binding file=prog to file=./bar.so.1: symbol bar
11753: .......
11753: transferring control: prog
11753: .......
11753: binding file=prog to file=./foo.so.1: symbol foo
11753: .......
```

Here, the symbol `bar`, which is required by a data relocation, is bound *before* the application gains control. Whereas the symbol `foo`, which is required by a function relocation, is bound *after* the application gains control when the

function is first called. This demonstrates the default mode of lazy binding. If the environment variable `LD_BIND_NOW` is set, all symbol bindings will occur before the application gains control.

Additional information regarding the real, and relative addresses of the actual binding locations can be obtained by setting `LD_DEBUG=bindings,detail`.

When the runtime linker performs a function relocation it rewrites the *.plt* entry associated with the function so that any subsequent calls will go directly to the function. The environment variable `LD_BIND_NOT` can be set to any value to prevent this *.plt* update. By using this variable together with the debugging request for detailed bindings, you can get a complete runtime account of all function binding. The output from this combination can be excessive, and the performance of the application will be degraded.

Another aspect of the runtime environment that can be displayed involves the various search paths used. For example, the search path mechanism used to locate any shared library dependencies can be displayed by setting `LD_DEBUG=libs`:

```
$ LD_DEBUG=libs prog
11775:
11775: find library=foo.so.1; searching
11775:  search path=/tmp:.  (RPATH from file prog)
11775:  trying path=/tmp/foo.so.1
11775:  trying path=./foo.so.1
11775:
11775: find library=bar.so.1; searching
11775:  search path=/tmp:.  (RPATH from file prog)
11775:  trying path=/tmp/bar.so.1
11775:  trying path=./bar.so.1
11775: .......
```

Here, the runpath recorded in the application `prog` affects the search for the two dependencies `foo.so.1` and `bar.so.1`.

In a similar manner, the search paths of each symbol lookup can be displayed by setting `LD_DEBUG=symbols`. If this is combined with a `bindings` request, a complete picture of the symbol relocation process can be obtained:

```
$ LD_DEBUG=bindings,symbols
11782: .......
11782: symbol=bar;  lookup in file=./foo.so.1  [ ELF ]
11782: symbol=bar;  lookup in file=./bar.so.1  [ ELF ]
11782: binding file=prog to file=./bar.so.1: symbol bar
11782: .......
11782: transferring control: prog
11782: .......
11782: symbol=foo;  lookup in file=prog  [ ELF ]
11782: symbol=foo;  lookup in file=./foo.so.1  [ ELF ]
11782: binding file=prog to file=./foo.so.1: symbol foo
11782: .......
```

**Note –** In the previous example the symbol `bar` is not searched for in the application `prog`. This is due to an optimization used when processing copy relocations (see “Profiling Shared Objects” on page 111 for more details of this relocation type).

# 3