Dynamic Linking 1

The SunOS operating system has long provided efficient memory and disk usage through *dynamic linking*. Dynamic linking allows applications to share code by enabling them to link to shared objects at runtime. A shared object can be a standard library such as `libc.so` or a custom library.

By separating an application from the services of standard libraries, dynamic linking also increases the portability and extensibility of an application. This separation between the interface of a service and its implementation enables the system to evolve while maintaining application stability which is crucial for conformance with the Application Binary Interface (ABI). Thus, dynamic linking is the preferred compilation method for Solaris applications.

The strategy for dynamic linking in Sun OS 5.x is similar to that in SunOS 4.1.x. This chapter provides an overview of dynamic linking concepts and describes the changes made in dynamic linking between SunOS 4.1.x and SunOS 5.x.

What is Dynamic Linking

To grasp dynamic linking and its benefits, it is useful to understand the linking process and the other mode of linking: *static linking*.

Linking Overview

Linking, or *link editing*, is the set of operations necessary to build an executable program from one or more object files. During linking, symbols referenced in one module are bound with definitions from other modules. More concretely, linking is the process by which the symbol `printf()` in the following sample source file, `hello.c`, is connected with its definition in the standard C library.

```
#include <stdio.h>
main()
{
    printf("hello, world\n");
}
```

The Link Editor

Linking begins with the link editor, `ld`, which takes one or more object modules and concatenates them into some form of output module. The link editor is commonly invoked as part of the compilation process (see `cc`), but it may be invoked directly. Although the link editor has numerous input options, the output module produced is one of four basic kinds as shown in Figure 1-1.

1. *Relocatable object*—a simple concatenation of input modules, which may be used in subsequent link edit phases.
2. *Static executable*—a ready-to-run process that has all symbolic references bound to the executable.
3. *Dynamic executable*—an executable that requires intervention by the runtime linker. Its symbolic references may still need to be bound at runtime, and it may have one or more dependencies in the form of shared objects.
4. *Shared object* (also referred to as *shared library*)—an output module which provides services that may be bound to a dynamic executable at runtime.

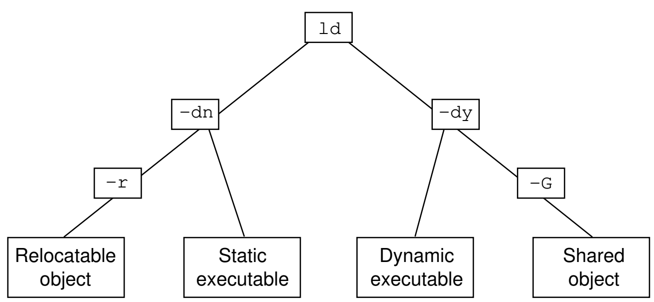

Figure 1-1 Static vs. Dynamic Linking at Link Edit Phase

Most applications are dynamic executables. Of the output modules produced by the link editor in Figure 1-1, the last two, *dynamic executables* and *shared objects*, are the main focus of this chapter.

File Format

The output files generated by the link editor contain information which identify its type. The contents of the output file are organized to adhere to a particular format.

Solaris 1.x produced and supported the `a.out` format which typically consisted of the following sections: header, program text, program data, text and data relocation information, a symbol table and a string table.

Solaris 2.x uses the more sophisticated *Executable and Linking Format* (ELF). A header at the beginning of the object file describes the file's organization. *Sections* hold the bulk of the object file information for the linking view: instructions, data symbol table, relocation information and so on. *Segments* hold the object file information for the program execution view.

ELF provides enhancements to `a.out` and other object file formats used in Solaris 2.x predecessors. Among these enhancements is information that eliminates the old `.sa` file complexities (see "Building Shared Objects" on page 1-66). Additionally, ELF provides the ability to extend the object file format to meet the requirements of the 1990's.

Two Modes of Linking

There are two modes of linking: static and dynamic. For both modes, the link editor searches each module of your program, including any libraries you have specified, for definitions of external symbols used but not defined in the other modules. If the link editor does not find a definition for a symbol, it reports an error by default and fails to create an executable program. The principal difference between static and dynamic linking lies in what happens after this search is completed.

Under *static linking*, copies of the archive object files that satisfy still unresolved external references in a program are incorporated in a program in the executable at link edit time. External references are connected with their definitions (assigned addresses in memory) when the executable is created. Under static linking, the link editor does not accept shared objects as input. Statically linked code cannot be shared, and may be duplicated any number of times in memory.

Under *dynamic linking*, the link editor reads the shared objects and performs symbol lookup and verification, but only records the name of the dependencies and some bookkeeping information in the executable. Hence, unlike static executables, a dynamic executable does not contain code for all the functions it calls. When building a dynamic executable, the link editor allows external symbols to remain unbound; these remaining symbols are left to be resolved by a runtime invocation of the link editor, `ld.so.x`. where `x` is the most recent version number. For example, on Solaris 2.1 the version of the runtime linker is `ld.so.1`.

The Runtime Linker

The *runtime linker* is responsible for processing dynamic executables and shared objects, and for combining them to create an executable process. At execution, before `main()` receives control, the runtime linker maps the contents of shared objects used by the process into the virtual address space of the process. The runtime linker first binds external data references with their definition. If it does not find a definition for a data symbol, the runtime linker reports an error and fails to create an executable program. For undefined external function references, the runtime linker, by default, resolves the reference upon the initial call to that function. Although it can be overridden, this “lazy binding” improves loading performance by eliminating bindings to functions that the program may never use.

The Benefits of Dynamic Linking

Dynamic linking should be used whenever possible. Most applications are dynamic executables which dynamically link to the standard shared libraries. The reasons applications should be dynamically linked include:

- *Resource efficiency*. Dynamically linked programs require less disk space since they do not contain copies of common library routines. They save system process memory by sharing library code at runtime. This also saves system overhead and I/O bandwidth by eliminating the need to page multiple copies of the same function between swap and RAM.
- *Portability*. A dynamically linked application can run without recompilation on different machines of the same machine architecture under different implementations of the ABI.
- *Stability*. Shared library code can be fixed or enhanced without having to relink the applications that use it. The runtime linker uses definitions from a new version of the shared library when resolving references to the shared code.
- *Flexibility*. As long as the pathname of a shared object is not hard-coded into an application, the shared object can be moved to a different directory without affecting that application.
- *Extensibility*. Dynamically linked programs can utilize the programmatic interfaces to dynamic linking, or can depend upon services which use those interfaces to extend themselves at execution time

Building Dynamic Executables

This section describes building a dynamic executable. Building shared objects are described later in this chapter. The man page for `ld` describes how to build the two other forms of link editor output: relocatable objects and static executables.

Using the Link Editor

The dynamic linking environment—made up of the link editor used in the dynamic mode and the runtime linker—provides a number of options for building an executable, based on the following principles:

1. Symbol definitions found in shared objects are not bound until runtime: the link editor verifies that the needed symbols are available from one or more shared objects, but their implementation can change by runtime, as long as their interfaces remains compatible.
2. The link editor and the runtime linker allow multiple definitions for needed symbols; the first one found wins.
3. Only the name of the shared objects containing the needed symbols must be committed to at link-edit time: their pathnames can be overwritten at runtime.

Linking with Additional Libraries

Compiler drivers automatically include one or more default libraries when building an executable. For example, `cc` includes the standard C library `libc.so`. Additional libraries to be included in link-editing must be supplied explicitly.

Library Naming Conventions

Libraries used by the link editor and runtime linker follow a naming convention. Each library is designated by the prefix `lib` and a suffix which is dependent on the library type. For a library, `foo`, the suffix naming convention in SunOS 5.x is as follows:

- `libfoo.so.`*x* - shared library with major version number of *x*. For example, `libc.so.1` is the shared object version of the standard C library.
- `libfoo.so` - typically the default version of a shared library implemented as a link to the most recent version. For example, `libfoo.so` would be a link to `libfoo.so.`*n* where *n* is the highest installed version.
- `libfoo.a` - archive file, (see `ar)`. For example, `libc.a` is the archive version of the standard C library.

You can provide libraries to the link-edit phase by using the `-l`*name* option where the *name* string is the library's filename without its prefix or suffix. By default, the link editor uses the `.so` version of the library. For example

```
$ cc file1.c file2.c file3.c -l foo
```

directs the link editor to search for `libfoo.so`, which would typically be a link to the most recent shared version of `foo`. If `libfoo.so` was not found, the link editor would search for `libfoo.a`.

Shared Library Version Numbering

As noted earlier in this section, a shared library file has a suffix that starts with `.so`, followed by a version number. In SunOS 4.1.x this version number consists of a major number followed by a minor number; in SunOS 5.x there is only a major number. The following discussion on versioning applies to major number versioning and uses SunOS 5.x examples.

Shared libraries use versioning to denote changes to the interface provided by that library. Hence, the version number distinguishes between incompatible releases of a shared library. A given installation may provide several versions of the library, enabling several generations of applications to run and preserving developers' investments in existing binary applications.

A system typically provides a default version of a library which usually corresponds to the highest installed version and is used in building applications. For example, the default C library used by the link editor is named via the path `/usr/lib/libc.so`. However, that path is implemented as a *link* to the current C library, `/usr/lib/libc.so.1`. If a new version of the C library, `/usr/lib/libc.so.2`, is installed, the site might choose to leave the first version as the default, or move the link from `/usr/lib/libc.so` to the new version. In either case, both versions would remain available so that programs built to use either of them would continue to run.

A shared library can be built to include its version information in the library. When a program is constructed using such a library, this version information is recorded in the resulting executable.The runtime linker uses this information to find the correct shared library to satisfy unresolved external references. If it cannot find the matching version, the runtime linker quits, terminating program execution.

Linking with a Mix of Shared Objects and Archives

When building a dynamic executable, you may choose to link with a mix of shared objects and archives.

To do so, toggle the `-B[static|dynamic]` option on the command line as many times as required. For example, to link your program with the archive `libfoo.a` and the shared object `libc.so`, issue the following command:

```
$ cc -o prog main.o file1.o file2.o file3.o -Bstatic -lfoo -Bdynamic
```

Directories to be Searched by the Link Editors

The link editor, `ld`, knows of only one standard place to look for libraries: `/usr/lib`. There are three methods to add directories to the link editor search path: setting the `LD_LIBRARY_PATH` environment variable, or using either the `-L` or the `-YP` command line option.

At link-edit time, `ld` searches for libraries in the directories you specify as follows:

- directories given in the `LD_LIBRARY_PATH` environment variable; this variable's value takes the form of a colon-separated list of pathnames
- each directory specified with a `-L` option to `cc`, the C compiler
- next, if `LD_LIBRARY_PATH` contains a semicolon, those directories listed after the semicolon
- finally, the standard directory for shared libraries (`/usr/lib`). The command line option, `-YP`, allows you to change the default directory.

In the following example, directories to be searched by the link editor have been specified by setting `LD_LIBRARY_PATH` and using the `-L` option.

The search order is

```
dirlist1:path1:path2...pathn:dirlist2:/usr/lib
```

as shown in the following example:

```
$ LD_LIBRARY_PATH=dirlist1;dirlist2
$ export LD_LIBRARY_PATH
$ cc -o prog main.o -Lpath1 ... -Lpath2 ...-Lpathn -lfoo
```

Directories to be Searched by the Runtime Linker

Similar to the link editor, the runtime linker knows of only one standard place to look for libraries: `/usr/lib`. To add other directories to this search, set the environment variables `LD_LIBRARY_PATH` or `LD_RUN_PATH`, or use the `-R` command line option.

The runtime linker searches for libraries in the directories you specify as follows:

- directories specified in the `LD_LIBRARY_PATH` environment variable. By modifying `LD_LIBRARY_PATH` at *execution time*, you can change the search path of the runtime linker.
- directories recorded with the application. By using the `-R` option or the `LD_RUN_PATH` environment variable at *build time* you can add to the search path of the runtime linker. Changing `LD_RUN_PATH` at execution time will not make the runtime linker search in the new places. If both `LD_RUN_PATH` and the `-R` option are specified, the `-R` option takes precedence.
- the standard directory for shared libraries (`/usr/lib`).

It is recommended that if your application uses libraries not in /usr/lib, invoke the -R option, rather than setting LD_LIBRARY_PATH. Using the LD_LIBRARY_PATH environment variable affects the runtime linker search path for all applications.

Building Shared Objects

The compilers, link editors and runtime linkers of SunOS 4.1.x and 5.x allow you to build shared objects. It is important to note that "shared objects" are shared only to the extent that they can be used without modification by the programs using them. The 4.1.x and 5.x compilers maximize the sharing of shared objects by supporting the generation of *position-independent code* (PIC). PIC allows the code to be placed at any process address without having to be relocated (modified) in any way, thus remaining shared.The shared libraries shipped with SunOS have always been built with code generated as PIC.

PIC generation techniques employed by compilers result in object code that uses a variety of *indirect references* that separate absolute addresses, which have to be set on a per-process basis, from the rest of the code. The result is the software uses less memory during execution. Specifically, the information modified during execution is represented as data (which is generally unique to each process), is collapsed to use less storage, and is localized to a small area of the address space. This minimizes the amount of per-process memory required during the execution of programs using the library.

SunOS 4.1.x

To build a .so file in SunOS 4.1.x, use the commands

```
$ cc -pic -c *.c
$ ld -o libname.so.ma.mi -assert pure-text *.o
```

where ma represents the major version number and mi represents the minor version number.

The compiler generates position independent code when the -pic option is set. The loader will verify that the library is in fact sharable when you set the option -assert pure-text in the command line.

In SunOS 4.1.x, any initialized data of a shared object is placed in a .sa file. The .sa file is an ar-format file (archive library, see ar) that is statically linked to the executable at compile time to ensure correct allocation.

You need to verify that your .c files do not contain definitions of external global initialized data; they may, however, contain references to such data. All such data definitions must be segregated into the .sa file. To build a .sa file:

1. Segregate the declaration of exported global initialized data from the sources for each object into separate source files.
2. Create a separate (static) archive library composed of these data objects and give it a name of the form `libname.sa.ma.mi`.
3. Use `ranlib` to place a symbol table in the `.sa` archive.

SunOS 5.x

Use the following commands to build a complete shared library in SunOS 5.x. There is no need to build a `.sa` file.

```
$ cc -c -K PIC *.c
$ cc -G -z text -o libname.so.ma *.o
```

When the `-K PIC` option is set, the compiler generates position-independent code. The `-z text` option is the SunOS 5.x version of the SunOS 4.1.x assert pure-text option.

Including Version Information

As noted earlier, a shared object can be built to include its own version information. In general, the internal version information is the name of the library without any directory path information. The absence of any directory path information allows the library search path mechanisms to be used to find the library.

The `-h` option to `cc` allows you to record a version name in a shared object. In the following command, `-h` stores the name given to it, `libfoo.so.1`, in the shared object itself.

```
$ cc -K PIC -h libfoo.so.1 -o libfoo.so.1 f1.c f2.c f3.c
```

When a program is constructed with a shared object that is internally identified with a versioned name, the version information is recorded in the resulting executable. In the next example, the `ln` command is used to create a link between the name `libfoo.so.1` and the name `libfoo.so`. The link editor searches for the latter name in the second command.

```
$ ln -s libfoo.so.1 libfoo.so
$ cc -o myprog ... -Ldir -lfoo
```

However, the link editor records in the executable, myprog, the versioned name given by -h, libfoo.so.1, rather than the name libfoo.so. When a subsequent, incompatible version of the library, libfoo.so.2 is installed, executables that depend on libfoo.so.1 will continue to link with it at runtime and execute properly.

Dynamic Linking Programming Interface

Solaris provides a programmatic interface to the dynamic linker which you can use to bind or unbind shared libraries during the course of program execution. This interface is useful when you are in the process of building a shared library for your application, and the start-up time for your application is a significant performance factor.

The dlopen() routine allows you to get access to a shared library, which is bound to the process's address space. dlsym() returns the address of a given symbol within a shared library. dlclose() deletes the reference to a shared object. When the last reference is deleted, the shared object is removed from the process's address space. dlerror() can be used to obtain information about the last error occurring as the result of dlopen(), dlsym() or dlclose().

The program in Code Example 1-1 illustrates the programmatic interface; the pnum() function is defined in the shared library libshared.so.1.

Code Example 1-1 Programmatic Interface Sample Program

```
/* libshared.so.1 */

# include <stdio.h>

...

void
pnum(int num)
{
    printf ("num = %d\n", num);
}
```

The program code follows:

Code Example 1-2 libshared.so.1

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

main()
{
```

Code Example 1-2 libshared.so.1 (Continued)

```
    void    *open_handle,  /* library handle */
        (*access_handle)();/* function handle */

    /*
     * Load and get a handle on our shared object.
     */
    open_handle = dlopen("./libshared.so.1", RTLD_LAZY);
    if (open_handle == NULL) {
        fprintf(stderr, "dlopen failed: %s\n",
            dlerror());
        exit(1);
    }

    /*
     * Obtain the symbol we wish to access (pnum).
     */
    access_handle = (void (*)())dlsym(open_handle, "pnum");
    if (access_handle == NULL) {
        fprintf(stderr, "dlsym failed: %s\n",
            dlerror());
        exit(1);
    }

    /*
     * Use the handle to call our function.
     */
    (*access_handle)(10);

    /*
     * Now, remove the shared object, and terminate.
     */
    if (dlclose(open_handle) != 0) {
        fprintf(stderr, "dlclose failed: %s\n",
            dlerror());
        exit (1);
    }
}
```

Increasing Shared Library Performance

You can enhance shared library performance by:

- minimizing the amount of writable storage in the library. The amount of such storage represents the “load” imposed on the system for each program using the library in terms of per-process storage and memory demands. Use automatic storage class variables in functions, or dynamically allocate static storage when a function is called. A rule-of-thumb that has proved worthwhile in building the shared C library is any static storage larger than about 20 bytes should be dynamically allocated via `malloc()` when functions that use it are first called. This strategy works particularly well if a library consists of many families of functions, not all of which are likely to be used in all programs accessing the library.
- using functional interfaces rather than global variables whenever possible. The performance benefit of this is the use of the storage can demand its allocation, just as with the static storage described previously. A further functional benefit is the interface presented to the library can be made more stable by not relying on the shape of the storage used to implement global data.
- making every effort to minimize the memory load imposed by the library. Having a shared library can be an effective way to do this. However, it must be noted that *all* of a shared library is used when any of it is accessed. Therefore, it may be important to ensure that developers using the library incur the minimal amount of memory load to use a given function. This can be accomplished by putting families of related functions in the same place in the library. A common problem is the inclusion of logically read-only storage in the library’s writable data segment. Such storage fragments the storage which is actually written, spreading it over more physical memory pages than might otherwise be necessary, and increasing the memory load. Using `const`, compiler switches, or other means to move read-only data to the actual read-only parts of the library is highly recommended.
- using analysis tools to automate the process of increasing a program’s (or shared object’s) “locality”. Many manufacturers provide such tools. Wherever possible, use of these in the creation of production shared objects is highly recommended.

Dynamic Linking and the Binary Compatibility Package

Solaris 2.x provides an optional *Binary Compatibility Package* (BCP) which supports the execution of pre-Solaris 2.x applications. These applications, which are not in the ELF format, can be run without recompilation, provided they meet certain requirements.

The precise requirements will vary based on the instruction set of the Solaris 2.x computer. To execute in BCP (SUNWbcp) mode on a Solaris 2.x SPARC computer, a pre-Solaris 2.x application:

- must be dynamically linked
- must not access `libkvm` or `/dev/kmem`
- must not write directly to system files

- must not rely on customer-supplied drivers or ioctls
- must not trap directly into the kernel
- must use only publicized SunOS interfaces

Applications meeting these conditions can run simultaneously with "new" applications when the BCP is installed. No special action is required to run such applications—the system employs the compatibility mechanisms transparently.

The BCP facilitates the transition to Solaris 2.x by allowing you to run some of your older applications which have not yet been ported to Solaris 2.x. However, the development and use of native 2.x applications is strongly encouraged due to the following limitations of the BCP:

- *Performance*. The BCP application will generally run slower than its native counterpart. Additionally, the BCP environment can impose extra system load in an environment running a mix of BCP and native applications.
- *Debugging*. The ability to diagnose and correct problems in BCP applications is quite limited.
- *Future compatibility*. The underlying mechanisms that enable support of some older applications may eventually be obsoleted.
- *Features*. The new capabilities of the Solaris 2.x will not be available to BCP applications.

The transition to native 2.x applications is encouraged at your earliest convenience.

For more information on the compatibility package, read the *SunOS 5.0 Binary Compatibility Guide*.

Bibliography

"Shared Libraries", Robert A. Gingell, *UNIX Review*, August 1989, Volume 7, Number 8.

"Shared Libraries in SunOS", Robert A. Gingell, et al, Sun Microsystems Inc., Proceedings of 1987 Summer USENIX Conference, USENIX Association.

SunOS 5.0 Linker and Libraries Manual, Part Number 800-6873-06, SunSoft, Inc., Mountain View. CA.

Programmer's Overview, Part Number 800-3847-10, Sun Microsystems Inc., Mountain View, CA.

System V Application Binary Interface: SPARC Processor Supplement, UNIX System Laboratories, Prentice Hall, Englewood Cliffs, NJ.

System V ANSI C and Programming Support Tools, UNIX System Laboratories, Prentice Hall, Englewood Cliffs, NJ.