Shared Objects 4

Overview

Shared objects are one form of output created by the link-editor, and are generated by specifying the `-G` option. For example:

```
$ cc -o libfoo.so.1 -G -K pic foo.c
```

Here the shared object `libfoo.so.1` is generated from the input file `foo.c`.

Note – This is a simplified example of generating a shared object. Usually, additional options are recommended, and these will be discussed in subsequent sections of this chapter.

A shared object is an *indivisible* unit generated from one or more relocatable objects. Shared objects can be bound with dynamic executables to form a runable process. As their name implies, shared objects can be *shared* by more than one application. Because of this potentially far-reaching effect, this chapter describes this form of link-editor output in greater depth than has been covered in previous chapters.

4

For a shared object to be bound to a dynamic executable or another shared object, it must first be available to the link-edit of the required output file. During this link-edit, any input shared objects are interpreted as if they had been added to the logical address space of the output file being produced. That is, *all* the functionality of the shared object is made available to the output file.

These shared objects become *dependencies* of this output file. A small amount of bookkeeping information is maintained within the output file to describe these dependencies. The runtime linker interprets this information and completes the processing of these shared objects as part of creating a runable process.

The following sections expand upon the use of shared objects within the *compilation* and *runtime* environments (these environments are introduced in "Shared Objects" on page 4). Issues that complement and help coordinate the use of shared objects within these environments are covered, together with techniques that maximize the efficiency of shared objects.

Naming Conventions

Neither the link-editor, nor the runtime linker, interprets any file by virtue of its filename. All files are inspected to determine their ELF type (see "ELF Header" on page 142). From this information the processing requirements of the file are deduced. However, shared objects usually follow one of two naming conventions depending on whether they are being used as part of the compilation environment or the run-time environment.

When used as part of the compilation environment, shared objects are read and processed by the link-editor. Although these shared objects can be specified by explicit filenames as part of the command-line passed to the link-editor, it is more common that the `-l` option be used to take advantage of the link-editor's library search capabilities (see "Shared Object Processing" on page 13).

For a shared object to be applicable to this link-editor processing it should be designated with the prefix `lib` and the suffix `.so`. For example, `/usr/lib/libc.so` is the shared object representation of the standard C library made available to the compilation environment.

When used as part of the runtime environment, shared objects are read and processed by the runtime linker. Here it might be necessary to allow for change in the exported interface of the shared object over a series of software releases. This interface change can be anticipated and supported by providing the shared object as a *versioned* filename.

A versioned filename commonly takes the form of a `.so` suffix followed by a version number. For example, `/usr/lib/libc.so.1` is the shared object representation of version *one* of the standard C library made available to the runtime environment.

If a shared object is never intended for use within a compilation environment its name might drop the conventional `lib` prefix. Examples of shared objects that fall into this category are those used solely with `dlopen(3X)`. A suffix of `.so` is still recommended to indicate the actual file type, and a version number is strongly recommended to provide for the correct binding of the shared object across a series of software releases.

Note – The shared object name used in a `dlopen(3X)` is usually represented as a *simple* filename - in other words there is no '`/`' in the name. This convention provides flexibility by allowing the runtime linker to use a set of rules to locate the actual file (see "Loading Additional Objects" on page 62 for more details).

In Chapter 5, "Versioning", the concept of versioning a shared objects interface over a series of software releases is described in more detail. In addition, a mechanism for coordinating the naming conventions between shared objects used in both the compilation and runtime environments is presented. But first, a mechanism that allows a shared object to record its own runtime name is described.

Recording a Shared Object Name

The recording of a *dependency* in a dynamic executable or shared object will, by default, be the filename of the associated shared object as it is referenced by the link-editor. For example, the following dynamic executables, built against the same shared object `libfoo.so`, result in different interpretations of the same dependency:

```
$ cc -o ../tmp/libfoo.so -G foo.o
$ cc -o prog main.o -L../tmp -lfoo
$ dump -Lv prog | grep NEEDED
[1]     NEEDED   libfoo.so

$ cc -o prog main.o ../tmp/libfoo.so
$ dump -Lv prog | grep NEEDED
[1]     NEEDED   ../tmp/libfoo.so

$ cc -o prog main.o /usr/tmp/libfoo.so
$ dump -Lv prog | grep NEEDED
[1]     NEEDED   /usr/tmp/libfoo.so
```

As these examples show, this mechanism of recording dependencies can result in inconsistencies due to different compilation techniques. Also, the location of a shared object as referenced during the link-edit might differ from the eventual location of the shared object on an installed system.

To provide a more consistent means of specifying dependencies, shared objects can record within themselves the filename by which they should be referenced at runtime.

During the link-edit of a shared object, its runtime name can be recorded within the shared object itself by using the `-h` option. For example:

```
$ cc -o ../tmp/libfoo.so -G -K pic -h libfoo.so.1 foo.c
```

Here, the shared object's runtime name `libfoo.so.1`, is recorded within the file itself. This identification is known as an *soname*, and its recording can be displayed using `dump(1)` and referring to the entry that has the `SONAME` tag. For example:

```
$ dump -Lvp ../tmp/libfoo.so

../tmp/libfoo.so:
[INDEX] Tag      Value
[1]     SONAME   libfoo.so.1
.........
```

When the link-editor processes a shared object that contains an *soname*, it is this name that is recorded as a dependency within the output file being generated.

Therefore, if this new version of `libfoo.so` is used during the creation of the dynamic executable `prog` from the previous example, all three methods of building the executable result in the same dependency recording:

```
$ cc -o prog main.o -L../tmp -lfoo
$ dump -Lv prog | grep NEEDED
[1]     NEEDED   libfoo.so.1

$ cc -o prog main.o ../tmp/libfoo.so
$ dump -Lv prog | grep NEEDED
[1]     NEEDED   libfoo.so.1

$ cc -o prog main.o /usr/tmp/libfoo.so
$ dump -Lv prog | grep NEEDED
[1]     NEEDED   libfoo.so.1
```

In the examples shown above, the `-h` option is used to specify a *simple* filename - in other words there is no '/' in the name. This convention is recommended, as it provides flexibility by allowing the runtime linker to use a set of rules to locate the actual file (see "Locating Shared Object Dependencies" on page 54 for more details).

Inclusion of Shared Objects in Archives

The mechanism of recording an *soname* within a shared object is essential if the shared object is ever processed from an archive library.

An archive can be built from one or more shared objects and then used to generate a dynamic executable or shared object. Shared objects can be extracted from the archive to satisfy the requirements of the link-edit (see "Archive Processing" on page 12 for more details on the criteria for archive extraction). However, unlike the processing of relocatable objects, which are concatenated to the output file being created, any shared objects extracted from the archive will be recorded as dependencies.

The name of an archive member is constructed by the link-editor and is a concatenation of the archive name and the object within the archive. For example:

```
$ cc -o libfoo.so.1 -G -K pic foo.c
$ ar -r libfoo.a libfoo.so.1
$ cc -o main main.o libfoo.a
$ dump -Lv main | grep NEEDED
[1]     NEEDED   libfoo.a(libfoo.so.1)
```

As it is highly unlikely that a file with this concatenated name will exist at runtime, providing an *soname* within the shared object is the only means of generating a meaningful runtime filename for the dependency.

Note – The run-time linker does not extract objects from archives. Therefore, in the above example it will be necessary for the required shared object dependencies to be extracted from the archive and made available to the runtime environment.

Recorded Name Conflicts

When shared objects are used to build a dynamic executable or another shared object, the link-editor performs several consistency checks to insure that any dependency names that will be recorded in the output file are unique.

Conflicts in dependency names can occur if two shared objects used as input files to a link-edit both contain the same *soname*. For example:

```
$ cc -o libfoo.so -G -K pic -h libsame.so.1 foo.c
$ cc -o libbar.so -G -K pic -h libsame.so.1 bar.c
$ cc -o prog main.o -L. -lfoo -lbar
ld: fatal: file ./libbar.so: recording name `libsame.so.1' \
           matches that provided by file ./libfoo.so
ld: fatal: File processing errors.  No output written to prog
```

A similar error condition will occur if the filename of a shared object that does not have a recorded *soname* matches the *soname* of another shared object used during the same link-edit.

If the runtime name of a shared object being generated matches one of its dependencies the link-editor will also report a name conflict. For example:

```
$ cc -o libbar.so -G -K pic -h libsame.so.1 bar.c -L. -lfoo
ld: fatal: file ./libfoo.so: recording name `libsame.so.1'  \
           matches that supplied with -h option
ld: fatal: File processing errors.  No output written to libfoo.so
```

Shared Objects with Dependencies

Although most of the examples presented in this chapter so far have shown how shared object dependencies are maintained in dynamic executables, it is quite common for shared objects to have their own dependencies (this was introduced in "Shared Object Processing" on page 13).

In "Directories Searched by the Runtime Linker" on page 55, the search rules used by the runtime linker to locate shared object dependencies are covered. If a shared object does not reside in the default directory `/usr/lib`, then the

runtime linker must explicitly be told where to look. The preferred mechanism of indicating any requirement of this kind is to record a *runpath* in the object that has the dependencies by using the link-editor's `-R` option. For example:

```
$ cc -o libbar.so -G -K pic bar.c
$ cc -o libfoo.so -G -K pic foo.c -R/home/me/lib -L. -lbar
$ dump -Lv libfoo.so

libfoo.so:

  **** DYNAMIC SECTION INFORMATION ****
.dynamic :
[INDEX] Tag      Value
[1]     NEEDED   libbar.so
[2]     RPATH    /home/me/lib
.........
```

Here, the shared object `libfoo.so` has a dependency on `libbar.so`, which is expected to reside in the directory `/home/me/lib` at runtime.

It is the responsibility of the shared object to specify any runpath required to locate its dependencies. Any runpath specified in the dynamic executable will only be used to locate the dependencies of the dynamic executable, it will *not* be used to locate any dependencies of the shared objects.

However, the environment variable `LD_LIBRARY_PATH` has a more global scope, and any pathnames specified using this variable will be used by the runtime linker to search for *any* shared object dependencies. 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 (see "Directories Searched by the Runtime Linker" on page 55 for a more extensive discussion).

Dependency Ordering

In most of the examples in this document, dependencies of dynamic executables and shared objects are portrayed as unique and relatively simple (the breadth-first ordering of dependent shared objects is described in "Locating Shared Object Dependencies" on page 54). From these examples, the ordering of shared objects as they are brought into the process address space might seem very intuitive and predictable.

However, when dynamic executables and shared objects have dependencies on the same common shared objects, the order in which the objects are processed can become less predictable.

For example, assume a shared object developer generates `libfoo.so.1` with the following dependencies:

```
$ ldd libfoo.so.1
        libA.so.1 =>     ./libA.so.1
        libB.so.1 =>     ./libB.so.1
        libC.so.1 =>     ./libC.so.1
```

If you create a dynamic executable, `prog`, using this shared object, and also define an explicit dependency on `libC.so.1`, then the resulting shared object order will be:

```
$ cc -o prog main.c -R. -L.-lC -lfoo
$ ldd prog
        libC.so.1 =>     ./libC.so.1
        libfoo.so.1 =>   ./libfoo.so.1
        libA.so.1 =>     ./libA.so.1
        libB.so.1 =>     ./libB.so.1
```

Therefore, had the developer of the shared object `libfoo.so.1` placed a requirement on the order of processing of its dependencies, this requirement will be compromised by the construction of the dynamic executable `prog`.

Developers who place special emphasis on symbol interposition (see “Symbol Lookup” on page 59, “Symbol Lookup” on page 69 and “Using Interposition” on page 75), and *.init* section processing (see “Initialization and Termination Routines” on page 64), should be aware of this potential change in shared object processing order.

Shared Objects as Filters

A *filter* is a special form of shared object used to provide indirection to an alternative shared object. Two forms of shared object filter exist:

- a *standard* filter
- an *auxiliary* filter

A *standard* filter, in essence, consists solely of a symbol table, and provides a mechanism of *abstracting* the compilation environment from the runtime environment. A link-edit using the filter will reference the symbols provided by the filter itself, however the implementation to which the symbol references is provided from an alternative source at runtime.

Standard filters are identified using the link-edit's `-F` flag. This flag takes an associated filename indicating the shared object to be used to supply symbols at runtime. This shared object is referred to as the *filtee*.

If the filtee cannot be processed at runtime, or any symbol defined by the filter cannot be located within the filtee, a fatal runtime error will occur.

An *auxiliary* filter has a similar mechanism, however the filter itself contains an implementation corresponding to its symbols. A link-edit using the filter will reference the symbols provided by the filter itself, however the implementation can be provided from an alternative source at runtime.

Auxiliary filters are identified during the link-edit's `-f` flag. This flag takes an associated filename indicating the shared object which might be used to supply symbols at runtime. This shared object is referred to as the *filtee*

If the filtee cannot be processed at runtime, or any symbol defined by the filter cannot be located within the filtee, the value of the filter will be used.

Generating a Standard Filter

First let's define a filtee, `libbar.so.1`, on which this filter technology will be applied. This filtee might be built from several relocatable objects. One of these objects originates from the file `bar.c`, and supplies the symbols `foo` and `bar`:

```
$ cat bar.c
char * bar = "bar";

char * foo()
{
    return("defined in bar.c");
}
$ cc -o libbar.so.1 -G -K pic .... bar.c ....
```

A standard filter, `libfoo.so.1`, is generated for the symbols `foo` and `bar`, and indicates the association to the filtee `libbar.so.1`. For example:

```
$ cat foo.c
char * bar = 0;

char * foo(){}

$ LD_OPTIONS="-F libbar.so.1" \
    cc -o libfoo.so.1 -G -K pic -h libfoo.so.1 -R. foo.c
$ ln -s libfoo.so.1 libfoo.so
$ dump -Lv libfoo.so.1 | egrep "SONAME|FILTER"
[1]     SONAME   libfoo.so.1
[2]     FILTER   libbar.so.1
```

Note – Here the environment variable `LD_OPTIONS` is used to circumvent this compiler driver from interpreting the `-F` option as one of its own.

If the link-editor references the standard filter `libfoo.so.1` to build a dynamic executable or shared object, it will use the information from the filters symbol table during symbol resolution (see "Symbol Resolution" on page 21 for more details).

At runtime, any reference to the symbols of the filter will result in the additional loading of the filtee `libbar.so.1`. The runtime linker will use this filtee to resolve any symbols defined by `libfoo.so.1`.

For example, the following dynamic executable, `prog`, references the symbols `foo` and `bar` which are resolved during link-edit from the filter `libfoo.so.1`:

```
$ cat main.c
extern char * bar, * foo();

main(){
    (void) printf("foo() is %s: bar=%s\n", foo(), bar);
}
$ cc -o prog main.c -R. -L. -lfoo
$ prog
foo() is defined in bar.c: bar=bar
```

The execution of the dynamic executable `prog` results in the function `foo()`, and the data item `bar`, being obtained from the filtee `libbar.so.1`, *not* from the filter `libfoo.so.1`.

Note – In this example, the filtee `libbar.so.1` is uniquely associated to the filter `libfoo.so.1` and is not available to satisfy symbol lookup from any other objects that might be loaded as a consequence of executing `prog`.

Standard filters provide a mechanism for defining a subset interface of an existing shared object. This mechanism is used in Solaris to create the shared objects `/usr/lib/libsys.so.1` and `/usr/lib/libdl.so.1`. The former provides a subset of the standard C library `/usr/lib/libc.so.1`. This subset represents the ABI-conforming functions and data items that reside in the C library that *must* be imported by a conforming application.

The latter defines the user interface to the runtime linker itself. This interface provides an abstraction between the symbols referenced in a compilation environment (from `libdl.so.1`) and the actual implementation binding produced within the runtime environment (from `ld.so.1`).

As the code in a standard filter is never referenced at runtime, there is no point in adding content to any functions defined within the filter. Any filter code might require relocations, which will result in an unnecessary overhead when processing the filter at runtime. Functions are best defined as empty routines.

Care should also be taken when generating the data symbols within a filter. Data items should always be initialized to insure that they result in references from dynamic executables. Some of the more complex symbol resolutions carried out by the link-editor require knowledge of a symbol's attributes, including the symbols size (see "Symbol Resolution" on page 21 for more details).

Therefore, it is recommended that the symbols in the filter be generated so that their attributes match those of the symbols in the filtee. This insures that the link-editing process will analyze the filter in a manner compatible with the symbol definitions used at runtime.

Generating an Auxiliary Filter

The creation of an *auxiliary* filter is essentially the same as for a *standard* filter (see "Generating a Standard Filter" on page 92 for more details). First let's define a filtee, `libbar.so.1`, on which this filter technology will be applied. This filtee might be built from several relocatable objects. One of these objects originates from the file `bar.c`, and supplies the symbol `foo`:

```
$ cat bar.c
char * foo()
{
    return("defined in bar.c");
}
$ cc -o libbar.so.1 -G -K pic .... bar.c ....
```

A standard filter, `libfoo.so.1`, is generated for the symbols `foo` and `bar`, and indicates the association to the filtee `libbar.so.1`. For example:

```
$ cat foo.c
char * bar = "foo";

char * foo()
{
    return ("defined in foo.c");
}
$ LD_OPTIONS="-f libbar.so.1" \
    cc -o libfoo.so.1 -G -K pic -h libfoo.so.1 -R. foo.c
$ ln -s libfoo.so.1 libfoo.so
$ dump -Lv libfoo.so.1 | egrep "SONAME|AUXILIARY"
[1]     SONAME    libfoo.so.1
[2]     AUXILIARY libbar.so.1
```

Note – Here the environment variable `LD_OPTIONS` is used to circumvent this compiler driver from interpreting the `-f` option as one of its own.

If the link-editor references the auxiliary filter `libfoo.so.1` to build a dynamic executable or shared object, it will use the information from the filters symbol table during symbol resolution (see "Symbol Resolution" on page 21 for more details).

At runtime, any reference to the symbols of the filter will result in a search for the filtee `libbar.so.1`. If this filtee is found the runtime linker will use this filtee to resolve any symbols defined by `libfoo.so.1`. If the filtee was not found, or a symbol from the filter is not found in the filtee, then the original value of the symbol within the filter is used.

For example, the following dynamic executable, `prog`, references the symbols `foo` and `bar` which are resolved during link-edit from the filter `libfoo.so.1`:

```
$ cat main.c
extern char * bar, * foo();

main(){
    (void) printf("foo() is %s: bar=%s\n", foo(), bar);
}
$ cc -o prog main.c -R. -L. -lfoo
$ prog
foo() is defined in bar.c: bar=foo
```

The execution of the dynamic executable `prog` results in the function `foo()` being obtained from the filtee `libbar.so.1`, *not* from the filter `libfoo.so.1`. However, the data item `bar` is obtained from the filter `libfoo.so.1`, as this symbol has no alternative definition in the filtee `libbar.so.1`.

Auxiliary filters provide a mechanism for defining an alternative interface of an existing shared object. This mechanism is used in Solaris to provide optimized functionality within platform specific shared objects.

Performance Considerations

A shared object can be used by multiple applications within the same system. The performance of a shared object therefore can have far reaching effects, not only on the applications that use it, but on the system as a whole.

Although the actual code within a shared object will directly affect the performance of a running process, the performance issues focused upon here target the runtime processing of the shared object itself. The following sections investigate this processing in more detail by looking at aspects such as text size and purity, together with relocation overhead.

Useful Tools

Before discussing performance it is useful to be aware of some available tools and their use in analyzing the contents of an ELF file.

Frequently reference is made to the size of either the *sections* or the *segments* that are defined within an ELF file (for a complete description of the ELF format see Chapter 6, "Object Files"). The size of a file can be displayed using the `size(1)` command. For example:

```
$ size -x libfoo.so.1
59c + 10c + 20 = 0x6c8

$ size -xf libfoo.so.1
..... + 1c(.init) + ac(.text) + c(.fini) + 4(.rodata) + \
..... + 18(.data) + 20(.bss) .....
```

The first example indicates the size of the shared objects *text*, *data* and *bss*, a categorization that has traditionally been used throughout previous releases of the SunOS operating system. The ELF format provides a finer granularity for expressing data within a file by organizing the data into *sections*. The second example displays the size of each of the file's loadable sections.

Sections are allocated to units known as *segments*, some of which describe how portions of a file will be mapped into memory. These loadable segments can be displayed by using the `dump(1)` command and examining the `LOAD` entries. For example:

```
$ dump -ov libfoo.so.1

libfoo.so.1:
 ***** PROGRAM EXECUTION HEADER *****
Type         Offset       Vaddr        Paddr
Filesz       Memsz        Flags        Align

LOAD         0x94         0x94         0x0
0x59c        0x59c        r-x          0x10000

LOAD         0x630        0x10630      0x0
0x10c        0x12c        rwx          0x10000
```

Here, there are two segments in the shared object `libfoo.so.1`, commonly referred to as the *text* and *data* segments. The text segment is mapped to allow reading and execution of its contents (r-x), whereas the data segment is mapped to allow its contents to be modified (rwx). Notice that the memory size (*Memsz*) of the data segment differs from the file size (*Filesz*). This difference accounts for the *.bss* section, which is actually part of the data segment.

Programmers however, usually think of a file in terms of the symbols that define the functions and data elements within their code. These symbols can be displayed using `nm(1)`. For example:

```
$ nm -x libfoo.so.1

[Index]   Value      Size      Type  Bind  Other Shndx   Name
.........
[39]     |0x00000538|0x00000000|FUNC |GLOB |0x0  |7      |_init
[40]     |0x00000588|0x00000034|FUNC |GLOB |0x0  |8      |foo
[41]     |0x00000600|0x00000000|FUNC |GLOB |0x0  |9      |_fini
[42]     |0x00010688|0x00000010|OBJT |GLOB |0x0  |13     |data
[43]     |0x0001073c|0x00000020|OBJT |GLOB |0x0  |16     |bss
.........
```

The section that contains a symbol can be determined by referencing the section index (*Shndx*) field from the symbol table and by using `dump(1)` to display the sections within the file. For example:

```
$ dump -hv libfoo.so.1

libfoo.so.1:
           **** SECTION HEADER TABLE ****
[No]    Type    Flags   Addr      Offset    Size      Name
.........
[7]     PBIT    -AI     0x538     0x538     0x1c      .init

[8]     PBIT    -AI     0x554     0x554     0xac      .text

[9]     PBIT    -AI     0x600     0x600     0xc       .fini
.........
[13]    PBIT    WA-     0x10688   0x688     0x18      .data

[16]    NOBI    WA-     0x1073c   0x73c     0x20      .bss
.........
```

Using the output from both the previous `nm(1)` and `dump(1)` examples, the association of the functions `_init`, `foo` and `_fini` to the sections *.init*, *.text* and *.fini* can be seen. These sections, because of their read-only nature, are part of the text segment.

Similarly, it can be seen that the data arrays `data` and `bss` are associated with the sections *.data* and *.bss* respectively. These sections, because of their writable nature, are part of the data segment.

Note – The previous `dump(1)` display has been simplified for this example.

Armed with this tool information, you can analyze the location of code and data within any ELF file you generate. This knowledge will be useful when following the discussions in later sections.

4

The Underlying System

When an application is built using a shared object, the entire loadable contents of the object are mapped into the virtual address space of that process at run time. Each process that uses a shared object starts by referencing a single copy of the shared object in memory.

Relocations within the shared object are processed to bind symbolic references to their appropriate definitions. This results in the calculation of true virtual addresses which could not be derived at the time the shared object was generated by the link-editor. These relocations usually result in updates to entries within the process's data segment(s).

The memory management scheme underlying the dynamic linking of shared object's share memory among processes at the granularity of a page. Memory pages can be shared as long as they are not modified at runtime. If a process writes to a page of a shared object when writing a data item, or relocating a reference to a shared object, it generates a private copy of that page. This private copy will have no effect on other users of the shared object, however, this page will have lost any benefit of sharing between other processes. Text pages that become modified in this manner are referred to as *impure*.

The segments of a shared object that are mapped into memory fall into two basic categories; the *text* segment, which is read-only, and the *data* segment which is read-write (see "Useful Tools" on page 97 on how to obtain this information from an ELF file). An overriding goal when developing a shared object is to maximize the text segment and minimize the data segment. This optimizes the amount of code sharing while reducing the amount of processing needed to initialize and use a shared object. The following sections present mechanisms that can help achieve this goal.

Position-Independent Code

To create programs that require the smallest amount of page modification at run time, the compiler will generate position-independent code under the `-K pic` option. Whereas the code within a dynamic executable is usually tied to a fixed address in memory, position-independent code can be loaded anywhere in the address space of a process. Because the code is not tied to a specific address, it will execute correctly without page modification at a different address in each process that uses it.

When you use position-independent code, relocatable references are generated in the form of an indirection which will use data in the shared object's data segment. The result is that the text segment code will remain read-only, and all relocation updates will be applied to corresponding entries within the data segment. See "Global Offset Table (Processor-Specific)" on page 213, "Procedure Linkage Table" on page 215 and "Procedure Linkage Table" on page 218 for more details on the use of these two sections.

If a shared object is built from code that is not position-independent, the text segment will usually require a large number of relocations to be performed at runtime. Although the runtime linker is equipped to handle this, the system overhead this creates can cause serious performance degradation.

A shared object that requires relocations against its text segment can be identified by using `dump(1)` and inspecting the output for any `TEXTREL` entry. For example:

```
$ cc -o libfoo.so.1 -G -R. foo.c
$ dump -Lv libfoo.so.1 | grep TEXTREL
[9]     TEXTREL  0
```

Note – The value of the `TEXTREL` entry is irrelevant, its presence in a shared object indicates that text relocations exist.

A recommended practice to prevent the creation of a shared object that contains text relocations is to use the link-editor's `-z text` flag. This flag causes the link-editor to generate diagnostics indicating the source of any non position-independent code used as input, and results in a failure to generate the intended shared object. For example:

```
$ cc -o libfoo.so.1 -z text -G -R. foo.c
Text relocation remains                         referenced
    against symbol                  offset      in file
foo                                 0x0         foo.o
bar                                 0x8         foo.o
ld: fatal: relocations remain against allocatable but non-
writable sections
```

Here, two relocations are generated against the text segment because of the non-position-independent code generated from the file `foo.o`. Where possible, these diagnostics will indicate any symbolic references that are required to carry out the relocations. In this case the relocations are against the symbols `foo` and `bar`.

Besides not using the `-K pic` option, the most common cause of creating text relocations when generating a shared object is by including hand written assembler code that has not been coded with the appropriate position-independent prototypes.

Note – By using the compiler's ability to generate an intermediate assembler file, the coding techniques used to enable position-independence can usually be revealed by experimenting with some simple test case source files.

A second form of the position-independence flag, `-K PIC`, is also available on some processors, and provides for a larger number of relocations to be processed at the cost of some additional code overhead (see `cc(1)` for more details).

Maximizing Shareability

As mentioned in "The Underlying System" on page 100, only a shared object's text segment is shared by all processes that use it, its data segment typically is not. Each process that uses a shared object usually generates a private memory copy of its entire data segment as data items within the segment are written to. A goal is to reduce the data segment, either by moving data elements that will never be written to the text segment, or by removing the data items completely.

The following sections cover several mechanisms that can be used to reduce the size of the data segment.

Move Read-Only Data to Text

Any data elements that are read-only should be moved into the text segment. This can be achieved using `const` declarations. For example, the following character string will reside in the *.data* section, which is part of the writable data segment:

```
char * rdstr = "this is a read-only string";
```

whereas, the following character string will reside in the *.rodata* section, which is the read-only data section contained within the text segment:

```
const char * rdstr = "this is a read-only string";
```

Although reducing the data segment by moving read-only elements into the text segment is an admirable goal, moving data elements that require relocations can be counter productive. For example, given the array of strings:

```
char * rdstrs[] = { "this is a read-only string",
                    "this is another read-only string" };
```

it might at first seem that a better definition is:

```
const char * const rdstrs[] = { ..... };
```

thereby insuring that the strings and the array of pointers to these strings are placed in a *.rodata* section. The problem with this definition is that even though the user perceives the array of addresses as read-only, these addresses must be relocated at runtime. This definition will therefore result in the creation of text relocations. This definition is best represented as:

```
const char * rdstrs[] = { ..... };
```

so that the array strings are maintained in the read-only text segment, but the array pointers are maintained in the writable data segment where they can be safely relocated.

Note – Some compilers, when generating position-independent code, can detect read-only assignments that will result in runtime relocations, and will arrange for placing such items in writable segments (for example *.picdata*).

Collapse Multiply-Defined Data

Data can be reduced by collapsing multiply-defined data. A program with multiple occurrences of the same error messages can be better off by defining one global datum, and have all other instances reference this. For example:

```
const char * Errmsg = "prog: error encountered: %d";

foo()
{
    ......
    (void) fprintf(stderr, Errmsg, error);
    ......
```

The main candidates for this sort of data reduction are strings. String usage in a shared object can be investigated using `strings(1)`. For example:

```
$ strings -10 libfoo.so.1 | sort | uniq -c | sort -rn
```

will generate a sorted list of the data strings within the file `libfoo.so.1`. Each entry in the list is prefixed with the number of occurrences of the string.

Use Automatic Variables

Permanent storage for data items can be removed entirely if the associated functionality can be designed to use automatic (stack) variables. Any removal of permanent storage will usually result in a corresponding reduction in the number of runtime relocations required.

Allocate Buffers Dynamically

Large data buffers should usually be allocated dynamically rather than being defined using permanent storage. Often this will result in an overall saving in memory, as only those buffers needed by the present invocation of an application will be allocated. Dynamic allocation also provides greater flexibility by allowing the buffer's size to change without effecting compatibility.

Minimizing Paging Activity

Many of the mechanisms discussed in the previous section "Maximizing Shareability" on page 102 will help reduce the amount of paging encountered when using shared objects. Here some additional generic software performance considerations are covered.

Any process that accesses a new page will cause a page fault. As this is an expensive operation, and because shared objects can be used by many processes, any reduction in the number of page faults generated by accessing a shared object will benefit the process and the system as a whole.

Organizing frequently used routines and their data to an adjacent set of pages will frequently improve performance because it improves the locality of reference. When a process calls one of these functions it might already be in memory because of its proximity to the other frequently used functions. Similarly, grouping interrelated functions will improve locality of references. For example, if every call to the function `foo()` results in a call to the function `bar()`, place these functions on the same page. Tools like `cflow(1)`, `tcov(1)`, `prof(1)` and `gprof(1)` are useful in determining code coverage and profiling.

It is also advisable to isolate related functionality to its own shared object. The standard C library has historically been built containing many unrelated functions, and only rarely, for example, will any single executable use everything in this library. Because of its widespread use, it is also somewhat difficult to determine what set of functions are really the most frequently used. In contrast, when designing a shared object from scratch it is better to maintain only related functions within the shared object. This will improve locality of reference and usually has the side effect of reducing the object's overall size.

4

Relocations

In "Relocation Processing" on page 57 the mechanisms by which the runtime linker relocates dynamic executables and shared objects to create a runable process was covered. "Symbol Lookup" on page 59, and "When Relocations are Performed" on page 60 categorized this relocation processing into two areas to simplify and help illustrate the mechanisms involved. These same two categorizations are also ideally suited for considering the performance impact of relocations.

Symbol Lookup

When the runtime linker needs to look 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 that the objects are mapped. In many instances the shared object that requires a symbolic relocation will turn out to be the provider of the symbol definition.

If this is the case, and the symbol used for this relocation is not required as part of the shared object's interface, then this symbol is a strong candidate for conversion to a *static* or *automatic* variable. A symbol reduction can also be applied to removed symbols from a shared objects interface (see "Reducing Symbol Scope" on page 38 for more details). By making these conversions the link-editor will incur the expense of processing any symbolic relocation against these symbols during the shared object's creation.

The only global data items that should be visible from a shared object are those that contribute to its user interface. However, frequently this is a hard goal to accomplish, as global data are often defined to allow reference from two or more functions located in different source files. Nevertheless, any reduction in the number of global symbols exported from a shared object will result in lower relocation costs and an overall performance improvement.

When Relocations are Performed

All *data* reference relocations must be carried out during process initialization before the application gains control, whereas any *function* reference relocations can be *deferred* until the first instance of a function being called. By reducing the number of data relocations, the runtime initialization of a process will be reduced.

Initialization relocation costs can also be deferred by converting data relocations into function relocations, for example, by returning data items by a functional interface. This conversion usually results in a perceived performance improvement as the initialization relocation costs are effectively spread throughout the process's lifetime. It is also possible that some of the functional interfaces will never be called by a particular invocation of a process, thus removing their relocation overhead altogether.

The advantage of using a functional interface can be seen in the next section "Copy Relocations". This section examines a special, and somewhat expensive, relocation mechanism employed between dynamic executables and shared objects, and provides an example of how this relocation overhead can be avoided.

Copy Relocations

Shared objects are usually built with position-independent code. References to external data items from code of this type employs indirect addressing through a set of tables (see "Position-Independent Code" on page 100 for more details). These tables are updated at runtime with the real address of the data items, which allows access to the data without the code itself being modified.

Dynamic executables however, are generally not created from position-independent code. Therefore it would seem that any references to external data they make can only be achieved at runtime by modifying the code that makes the reference. Modifying any text segment is something to be avoided, and so a relocation technique is employed to solve this reference which is known as a *copy* relocation.

When the link-editor is used to build a dynamic executable, and a reference to a data item is found to reside in one of the dependent shared objects, space is allocated in the dynamic executable's *.bss*, equivalent in size to the data item found in the shared object. This space is also assigned the same symbolic name as defined in the shared object. Along with this data allocation, the link-editor generates a special copy relocation record that will instruct the runtime linker to copy the data from the shared object to this allocated space within the dynamic executable.

Because the symbol assigned to this space is global, it will be used to satisfy any references from any shared objects. The effect of this is that the dynamic executable inherits the data item, and any other objects within the process that make reference to this item will be bound to this copy. The original data from which the copy is made effectively becomes unused.

This mechanism is best explained with an example. This example uses an array of system error messages that is maintained within the standard C library. In previous SunOS operating system releases, the interface to this information was provided by two global variables, `sys_errlist[]`, and `sys_nerr`. The first variable provided the array of error message strings, while the second conveyed the size of the array itself. These variables were commonly used within an application in the following manner:

```
$ cat foo.c
extern int      sys_nerr;
extern char *   sys_errlist[];

char *
error(int errnumb)
{
        if ((errnumb < 0) || (errnumb >= sys_nerr))
                return (0);
        return (sys_errlist[errnumb]);
}
```

Here the application is using the function `error` to provide a focal point to obtain the system error message associated with the number `errnumb`.

Examining a dynamic executable built using this code shows the implementation of the copy relocation in more detail:

```
$ cc -o prog main.c foo.c
$ nm -x prog | grep sys_
[36]  |0x00020910|0x00000260|OBJT |WEAK |0x0  |16 |sys_errlist
[37]  |0x0002090c|0x00000004|OBJT |WEAK |0x0  |16 |sys_nerr
$ dump -hv prog | grep bss
[16]    NOBI    WA-    0x20908   0x908    0x268   .bss
$ dump -rv prog

    **** RELOCATION INFORMATION ****

.rela.bss:
Offset      Symndx                Type              Addend

0x2090c     sys_nerr              R_SPARC_COPY      0
0x20910     sys_errlist           R_SPARC_COPY      0
..........
```

Here the link-editor has allocated space in the dynamic executable's *.bss* to receive the data represented by `sys_errlist` and `sys_nerr`. These data will be copied from the C library by the runtime linker at process initialization. Thus, each application that uses these data will get a private copy of the data in its own data segment.

There are actually two downsides to this technique. First, each application pays a performance penalty for the overhead of copying the data at run time. Secondly, the size of the data array `sys_errlist` has now become part of the C library's interface. If the size of this array were to change, presumably as new error messages are added, any dynamic executables that reference this array have to undergo a new link-edit to be able to access any of the new error messages. Without this new link-edit, the allocated space within the dynamic executable is insufficient to hold the new data.

These drawbacks can be eliminated if the data required by a dynamic executable are provided by a functional interface. The ANSI C function `strerror(3C)` illustrates this point. This function is implemented such that it will return a pointer to the appropriate error string based on the error number supplied to it. One implementation of this function might be:

```
$ cat strerror.c
static const char * sys_errlist[] = {
        "Error 0",
        "Not owner",
        "No such file or directory",
        ......
};
static const int sys_nerr =
        sizeof (sys_errlist) / sizeof (char *);

char *
strerror(int errnum)
{
        if ((errnum < 0) || (errnum >= sys_nerr))
                return (0);
        return ((char *)sys_errlist[errnum]);
}
```

The error routine in `foo.c` can now be simplified to use this functional interface, which in turn will remove any need to perform the original copy relocations at process initialization.

Additionally, because the data are now local to the shared object the data are no longer part of its interface, which allows the shared object the flexibility of changing the data without adversely effecting any dynamic executables that use it. Eliminating data items from a shared object's interface will generally improve performance while making the shared object's interface and code easier to maintain.

Although copy relocations should be avoided, `ldd(1)`, when used with either the `-d` or `-r` options, can be used to verify any that exist within a dynamic executable.

For example, if the dynamic executable `prog` had originally been built against the shared object `libfoo.so.1` such that the following two copy relocations had been recorded:

```
$ nm -x prog | grep _size_
[36]    |0x000207d8|0x40|OBJT |GLOB |15   |_size_gets_smaller
[39]    |0x00020818|0x40|OBJT |GLOB |15   |_size_gets_larger
$ dump -rv size | grep _size_
0x207d8     _size_gets_smaller    R_SPARC_COPY      0
0x20818     _size_gets_larger     R_SPARC_COPY      0
```

and a new version of this shared object is supplied which contains different data sizes for these symbols:

```
$ nm -x libfoo.so.1 | grep _size_
[26]    |0x00010378|0x10|OBJT |GLOB |8    |_size_gets_smaller
[28]    |0x00010388|0x80|OBJT |GLOB |8    |_size_gets_larger
```

then running `ldd(1)` against the dynamic executable will reveal:

```
$ ldd -d prog
    libfoo.so.1 =>    ./libfoo.so.1
    ...........
    copy relocation sizes differ: _size_gets_smaller
       (file prog size=40; file ./libfoo.so.1 size=10);
      ./libfoo.so.1 size used; possible insufficient data copied
    copy relocation sizes differ: _size_gets_larger
       (file prog size=40; file ./libfoo.so.1 size=80);
       ./prog size used; possible data truncation
```

Here `ldd(1)` informs us that the dynamic executable will copy as much data as the shared object has to offer, but only accepts as much as its allocated space allows.

Profiling Shared Objects

The runtime linker is capable of generating profiling information for any shared objects processed during the running of an application. This is possible because the runtime linker is responsible for binding shared objects to an

application and is therefore able to intercept any *global* function bindings (these bindings take place through *.plt* entries - see "When Relocations are Performed" on page 60 for details of this mechanism).

The profiling of a shared object is enabled by specifying its name with the `LD_PROFILE` environment variable. You can analyze one shared object at a time using this environment variable. However, the setting of the environment variable can be used to analyze one or more applications use of the shared object. In the following example the use of `libc` by the single invocation of the command `ls(1)` is analyzed:

```
$ LD_PROFILE=libc.so.1  ls -l
```

In the following example the environment variable setting will cause any applications use of `libc` to accumulate the analyzed information for the duration that the environment variable is set:

```
$ LD_PROFILE=libc.so.1; export LD_PROFILE
$ ls -l
$ make
$ ...
```

When profiling is enabled, a profile data file is created, if it doesn't already exist, and is mapped by the runtime linker. In the above examples this data file is `/var/tmp/libc.so.1.profile`. You can also specify an alternative directory to store the profile data using the `LD_PROFILE_OUTPUT` environment variable.

This profile data file is used to deposit `profil(2)` data and call count information related to the specified shared objects use. This profiled data can be directly examined with `gprof(1)`.

Note – `gprof(1)` is most commonly used to analyze the `gmon.out` profile data created by an executable that has been compiled with the `-xpg` option of `cc(1)`. The runtime linkers profile analysis does *not* require any code to be compiled with this option. Applications whose dependent shared objects are being profiled should not make calls to `profil(2)`, because this system call does not provide for multiple invocations within the same process. For the

same reason, these applications must not be compiled with the `-xpg` option of `cc(1)`, as this compiler generated mechanism of profiling is also built on top of `profil(2)`.

One of the most powerful features of this profiling mechanism is to allow the analysis of a shared object as used by multiple applications. Frequently profiling analysis is carried out using one or two applications. However, a shared object, by its very nature, can be used by a multitude of applications. Analyzing how these applications use the shared object can offer insights into where energy might be spent to improvement the *overall* performance of the shared object.

The following example shows a performance analysis of `libc` over a build of several applications within a source hierarchy:

```
$ LD_PROFILE=libc.so.1 ; export LD_PROFILE
$ make
......
$ gprof -b /usr/lib/libc.so.1 /var/tmp/libc.so.1.profile

granularity: each sample hit covers 4 byte(s) ....

                                    called/total     parents
index  %time    self descendents  called+self    name      index
                                    called/total     children

.....
-----------------------------------------------
                0.33        0.00      52/29381     _gettxt [96]
                1.12        0.00     174/29381     _tzload [54]
               10.50        0.00    1634/29381     <external>
               16.14        0.00    2512/29381     _opendir [15]
              160.65        0.00   25009/29381     _endopen [3]
[2]     35.0  188.74        0.00   29381          _open [2]
-----------------------------------------------
.....
granularity: each sample hit covers 4 byte(s) ....

   %  cumulative     self              self    total
 time   seconds   seconds    calls  ms/call  ms/call name
 35.0     188.74   188.74    29381     6.42     6.42  _open [2]
 13.0     258.80    70.06    12094     5.79     5.79  _write [4]
  9.9     312.32    53.52    34303     1.56     1.56  _read [6]
  7.1     350.53    38.21     1177    32.46    32.46  _fork [9]
 ....
```

The special name <*external*> indicates a reference from outside of the address range of the shared object being profiled. Thus, in the above example, 1634 calls to the function `open(2)` within `libc` occurred from the dynamic executables, or from other shared objects, bound with `libc` while the profiling analysis was in progress.

Note – The profiling of shared objects is multi-threaded safe except in the case where one thread calls `fork(2)` while another thread is updating the profile data information. The use of `fork1(2)` removes this restriction.