Introduction 1

This manual describes the operations of the Solaris link-editor and runtime linker, together with the objects on which they operate. The basic operation of the Solaris linkers involves the combination of objects and the connection of symbolic references from one object to the symbolic definitions within another. This operation is often referred to as *binding*.

The main areas this manual expands upon are:

- The *Link-Editor*

  The link-editor, `ld(1)`, concatenates one or more input files (either relocatable objects, shared objects, or archive libraries) to produce one output file (either a relocatable object, an executable application, or a shared object). The link-editor is most commonly invoked as part of the compilation environment (see `cc(1)`).

- The *Runtime Linker*

  The runtime linker, `ld.so.1`[1], processes dynamic executables and shared objects at runtime, and binds them to create a runable process.

1. ld.so.1 is a special case of a shared object and therefore allows itself to be versioned. Here a version number of 1 is used, however later releases of Solaris might provide higher version numbers.

- *Shared Objects* (sometimes referred to as *Shared Libraries*)

  Shared objects are one form of output from the link-edit phase. However, their importance in creating a powerful, flexible runtime environment warrants a section of its own.

- *Object Files*

  The Solaris linkers work with files that conform to the *executable and linking format* (ELF).

These areas, although separable into individual topics, have a great deal of overlap. While explaining each area, this document brings together the connecting principles and designs.

Link-Editing

Link-editing takes a variety of input files, from `cc(1)`, `as(1)` or `ld(1)`, and concatenates and interprets the data within these input files to form a single output file. Although the link-editor provides numerous options, the output file produced is one of four basic types:

- *Relocatable object* – a concatenation of input relocatable objects, which can be used in subsequent link-edit phases.
- *Static executable* – a concatenation of input relocatable objects that has all symbolic references bound to the executable, and thus represents a ready-to-run process.
- *Dynamic executable* – a concatenation of input relocatable objects that requires intervention by the runtime linker to produce a runable process. Its symbolic references might still need to be bound at runtime, and it might have one or more dependencies in the form of shared objects.
- *Shared object* – a concatenation of input relocatable objects that provides services that might be bound to a dynamic executable at runtime. The shared object might also have dependencies on other shared objects.

These output files, and the key link-editor options used to create them, are shown in Figure 1-1 on page 3.

Dynamic executables and *shared objects*, are often referred to jointly as *dynamic objects*, and are the main focus of this document.

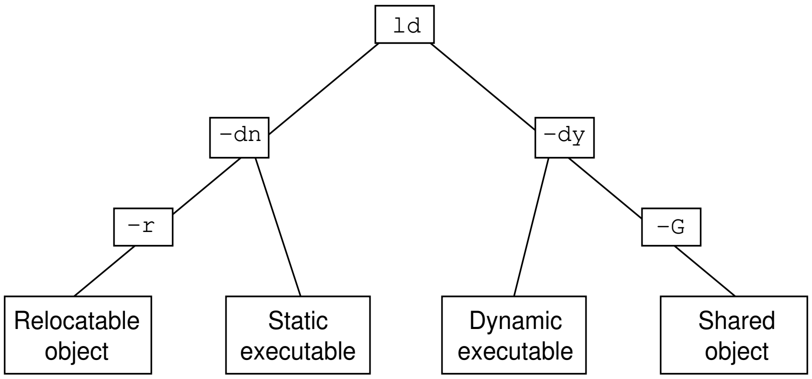

Figure 1-1 Static or Dynamic Link-editing

Runtime Linking

Runtime linking involves the binding of objects, usually generated from one or more previous link-edits, to generate a runable process. During the generation of these objects by the link-editor, the binding requirements are verified and appropriate bookkeeping information is added to each object to allow the runtime linker to map, relocate, and complete the binding process.

During the execution of the process, the facilities of the runtime linker are also made available and can be used to extend the process' address space by adding additional shared objects on demand. The two most common components involved in runtime linking are *dynamic executables* and *shared objects*.

Dynamic Executables

Dynamic executables are applications that are executed under the control of a runtime linker. These applications usually have dependencies in the form of shared objects, which are located and bound by the runtime linker to create a runable process. Dynamic executables are the default output file generated by the link-editor.

1

Shared Objects

Shared objects provide the key building block to a dynamically linked system. Basically, a shared object is similar to a dynamic executable, however shared objects usually have no entry point and they have not yet been assigned a virtual address.

Dynamic executables usually have dependencies on one or more shared objects. That is, the shared object(s) must be bound to the dynamic executable to produce a runable process. Because shared objects can be used by many applications, aspects of their construction directly affect shareability, versioning and performance.

It is useful to distinguish the processing of shared objects by either the link-editor or the runtime linker by referring to the *environments* in which the shared objects are being used:

- The *compilation* environment. Here, shared objects are processed by the link-editor to generate dynamic executables or other shared objects. The shared objects become dependencies of the output file being generated.
- The *runtime* environment. Here, shared objects are processed by the runtime linker, together with a dynamic executable, to produce a runable process.

Related Topics

Dynamic Linking

Dynamic linking is a term often used to embrace those portions of the link-editing process that generate dynamic executables and shared objects, together with the runtime linking of these objects to generate a runable process. Dynamic linking allows multiple applications to use the code provided by a shared object by enabling the application to bind to the shared object at runtime.

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, and is a crucial factor in providing an *application binary interface* (ABI). Dynamic linking is the preferred compilation method for Solaris applications.

Application Binary Interfaces

To enable the asynchronous evolution of system and application components, binary interfaces between these facilities are defined. The Solaris linkers operate upon these interfaces to assemble applications for execution. Although all components handled by the Solaris linkers have binary interfaces, one family of such interfaces of particular interest to applications writers is the *System V Application Binary Interface*.

The *System V Application Binary Interface*, or *ABI*, defines a system interface for compiled application programs. Its purpose is to document a standard binary interface for application programs on systems that implement the *System V Interface Definition, Third Edition*. Solaris provides for the generation and execution of ABI-conformant applications. On SPARC systems, the ABI is contained as a subset of the *SPARC® Compliance Definition* (SCD).

Many of the topics covered in the following chapters are influenced by the ABI. For more detailed information see the appropriate ABI manuals.

Support Tools

Together with the objects mentioned in the previous sections come several support tools and libraries. These tools provide for the analysis and inspection of these objects and the linking processes. Among these tools are: `nm(1)`, `dump(1)`, `ldd(1)`, `pvs(1)`, `elf(3E)`, and a linker debugging support library. Throughout this document many discussions are augmented with examples of these tools use.

1