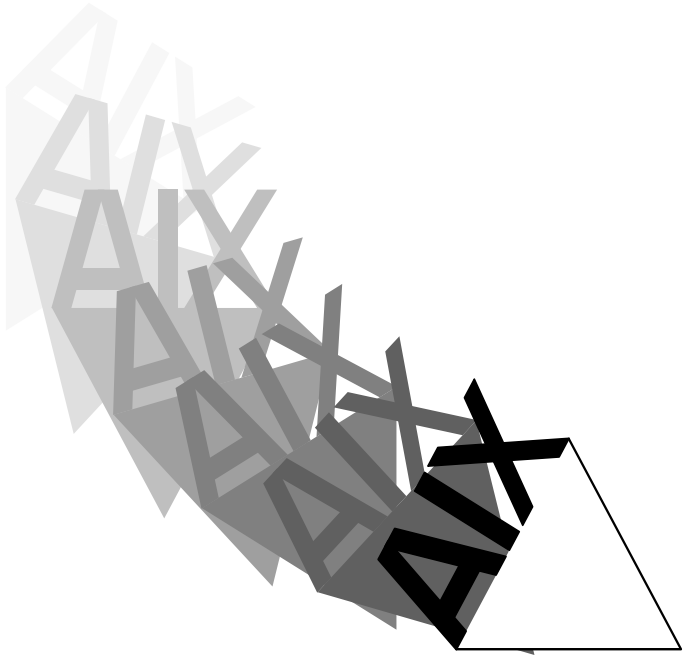
Linking and Binding

Gary R. Hook
hook@vnet.ibm.com



AIX Systems Center
Dallas, Texas

Overview

- ☞ Introduce and discuss the concepts and implications of dynamic linking and shared libraries
- ☞ Discuss differences between the AIX Version 3 linker and traditional UNIX linker implementations
- ☞ Investigate performance aspects of the linker
- ☞ Introduce the concepts of runtime loading and binding

Notes

AIX V3 supports shared objects, runtime symbol resolution, and dynamic binding to an executing process. It is this capability that, for example, allows the kernel to be dynamically extended while the system is running.

The AIX V3 linker utilizes different technology than previous UNIX implementations; this technology allows for dynamic modification of executing processes.

When designing a development methodology, the behavior of the linker should be considered.

It is possible for an executing program to easily load additional object modules and execute them. This is known as runtime binding.

Terms

☞ Linker

- ❏ the application program that combines multiple object modules into an executable program

☞ Loader

- ❏ a kernel function that accepts an executable file and maps it into the virtual memory of the system

☞ XCOFF Module

- ❏ a file containing machine code and one or more of: machine instructions, symbolic information, debugging information, initialized data, etc.

☞ Resolution

- ❏ the act of associating a reference to a symbol with the definition of that symbol

☞ Relocation

- ❏ the technique used to support the concept of address–independent programs

Notes

The command **ld** is the system linker. The linker processes command line arguments and hands instructions to the binder program **bind**. For the purposes of this document, the terms linker and binder are interchangeable.

The function of the loader is to prepare an object for execution by the system. This includes symbol resolution and virtual memory mapping.

When object modules refer to a symbol, that symbol must be resolved for proper execution to take place. The linker performs a portion of this function at link time; the loader handles runtime resolution when accessing shared objects.

AIX V3 supports address–independent, reentrant code. Text segments never contain address constants, allowing multiple processes to share a single copy of a text segment for an executing program.

Linking Phases

- ☞ Executable program construction using **ld**
- ☞ Dynamic binding performed by the system loader
- ☞ Runtime binding using system functions

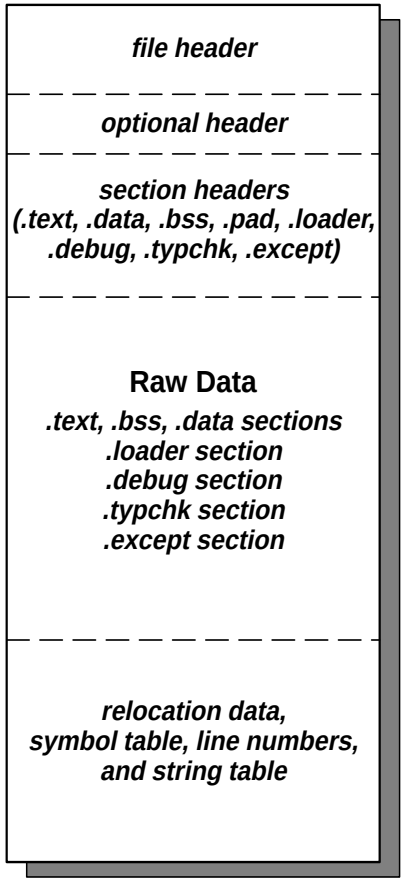
Notes

The **ld** command is traditionally used to construct an executable program. Much of the required symbol resolution is handled at this time.

When a program is loaded and prepared for execution, the system loader handles the resolution of symbols that refer to shared objects. This provides for the reuse of standard library routines in a transparent manner. Shared system library routines are stored in the Shared Text segment.

System calls are available which allow a running process to load and bind to an arbitrary object. The object is prepared by the linker for this purpose, but is accessed by some arbitrary executing process. This functionality causes loaded objects to be stored in the Process Private Data space instead of the Shared Text segment. Thus, application design should address this aspect when organizing executable code.

XCOFF Module

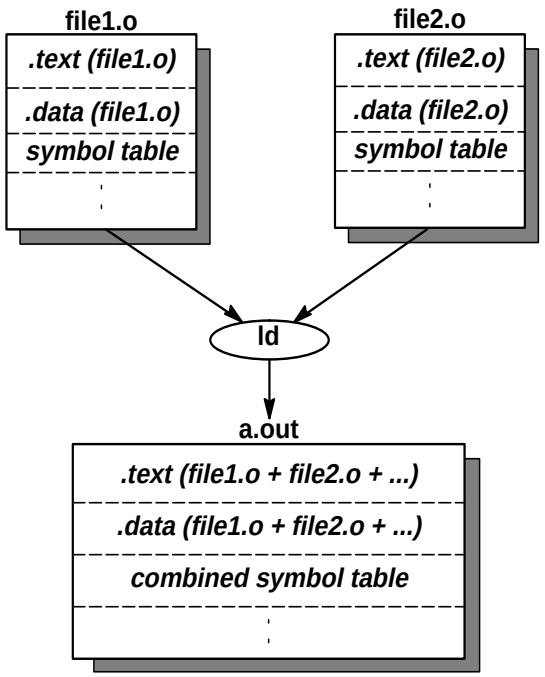


Notes

The object modules produced by a compiler must conform to a certain format so the system loader will be able to load and execute the program. Many UNIX systems use COFF (Common Object File Format) as the basis for their object module format. COFF consists of text (machine code), bss (uninitialized data), and data (initialized data) sections. In addition, COFF files contain a symbol table, relocation information, and a string table to store symbol names greater than eight characters in length. The COFF standard allows for additional sections to be added to provide for implementation–dependent extensions.

IBM uses the COFF standard format as the basis for their object module format, but they provide extensions to the standard to allow for shared libraries, dynamic linking, etc. IBM goes one step further by defining XCOFF as its object module format. XCOFF contains the same sections and information as COFF, but adds a loader section, a type check section, and an exception section. If debugging is specified by the compile command, a debug section is added along with line number information to keep track of valid breakpoints. This debug information is used by the various symbolic debuggers available. The loader section contains the information required by the dynamic loader. The type check section contains the parameter type–checking hash strings that are used to check for parameter type consistency at link or load time. The exception section is used to handle system exceptions.

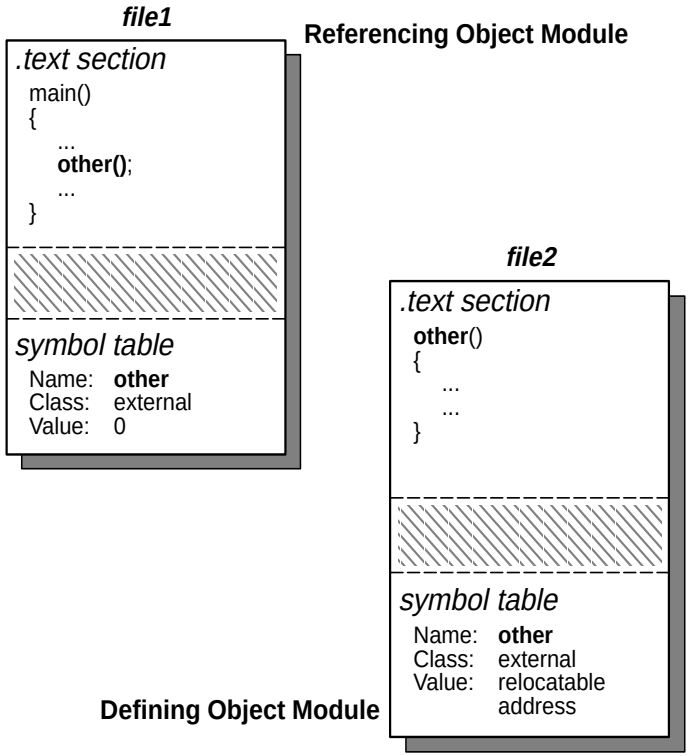
Combining Object Modules



Notes

The linker combines object modules (produced by compilers or the assembler) and resolves symbol references; the result is an executable object. The text sections from the object modules are combined to form one text section. The data sections from the object modules are also combined to form one data section. Symbol references are resolved when their definitions are encountered (resolution is discussed later), and the relocation information is updated to reflect the new addresses in the combined text and data sections (see Relocation).

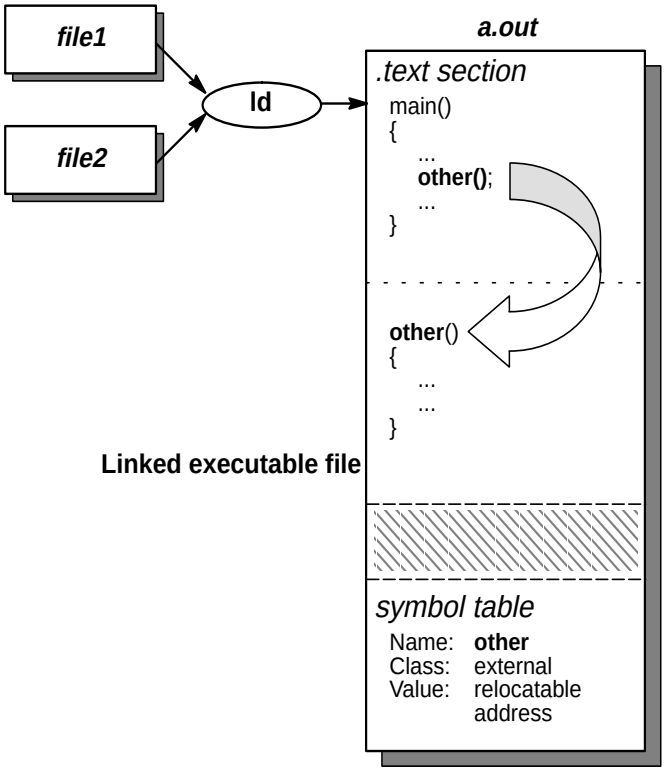
Linking Two Object Modules



Notes

As the text and data sections of the various object modules are being combined, resolution takes place to resolve external symbolic references. The address of an external data or subroutine reference is unknown; therefore, the machine code in an object module requires patching by the linker. External symbolic references result in the creation of relocation information (discussed in the following section), but the actual address of the symbolic reference, its definition, resides in a different object module. The object module containing the definition must be linked with the object module containing the symbolic reference. In other words, the linker matches external symbolic references with their appropriate definitions.

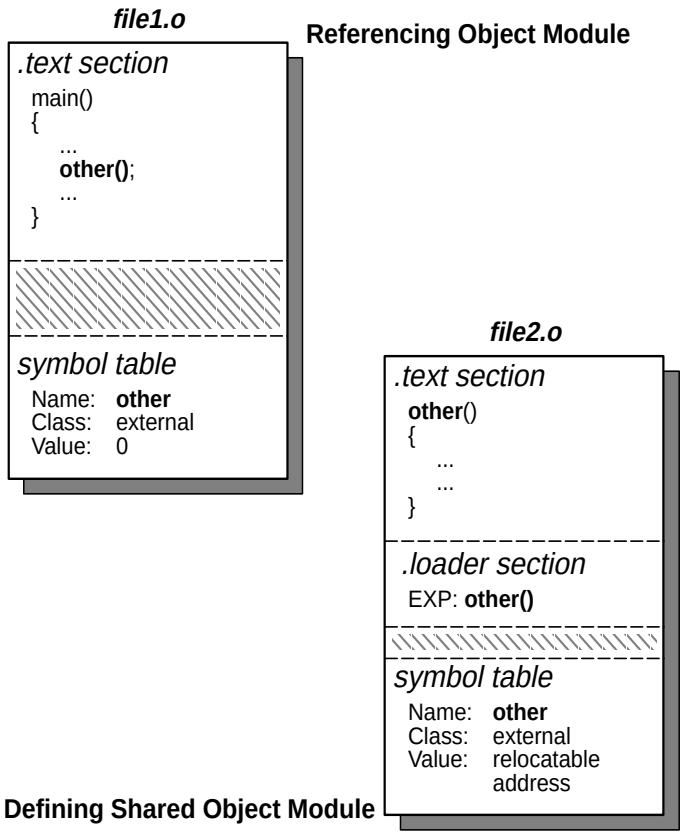
Linking Two Object Modules...



Notes

The text sections from the object modules are combined (along with any other text) and stored. During the resolution phase of the link, the linker creates one entry in the symbol table for the definition of **other**. The references to that symbol are then tied to the symbol table entry; the table contains the relocatable address for the function. Thus, a symbol reference can be viewed as an offset into the data segment to locate an address in the text segment; the address value is the only piece of information that must be updated at load time with a virtual location.

Linking to a Shared Object



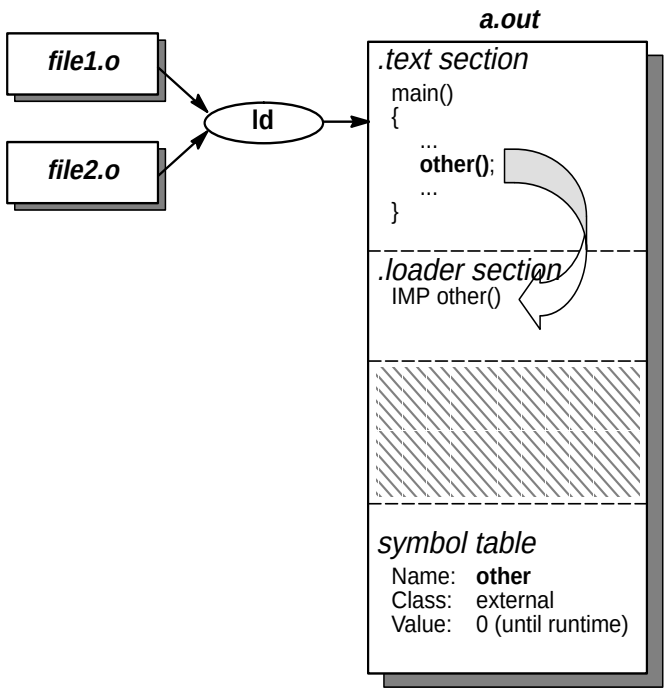
Notes

In AIX V3, unlike traditional UNIXs, resolution of library calls takes place at load time by default. IBM's XCOFF allows symbol resolution to be deferred until load time. That is, external symbolic references may be marked as IMPORTED in the loader section of the XCOFF module. When invoked, the loader brings in the object modules containing the definitions of these imported symbols and supplies the runtime addresses of these symbols to the referencing executable. If the object modules containing the symbol definitions in turn have imported symbols, then the object modules defining these symbols are loaded and these imported symbols are resolved. This continues until there are no unresolved imported symbols.

The object shown here containing the definition of the symbol of interest is organized in much the same fashion as in the previous example; the entry into the symbol table and the relocatable address exist here as well. The additional piece of information is the exported symbol in the loader section. This is required since the shared object will not ultimately be combined with the code that contains the symbol reference.

The linker recognizes the existence of the symbols in the loader section of the shared object, and utilizes this information to construct a new reference to the symbol. This new reference contains information necessary to locate the object module (containing the symbol definition) at load time. Thus, a portion of the symbol resolution process and code organization is deferred from link time to load time.

Linking to a Shared Object...



Dynamically Linked Executable File

Notes

Note that a symbol table entry is created in the linked application. This entry is similar to that in the statically linked version shown earlier. The key piece of information for the loader is the items marked for import in the loader section. The linker locates the definitions for these symbols in the appropriate object modules (this object module location information resides within the linked application) and modifies the symbol table entries to reflect their location. Since the symbol table must be written to, all address constants are stored in the data segment. The text segment is read–only, whereas the data segment is read–write.

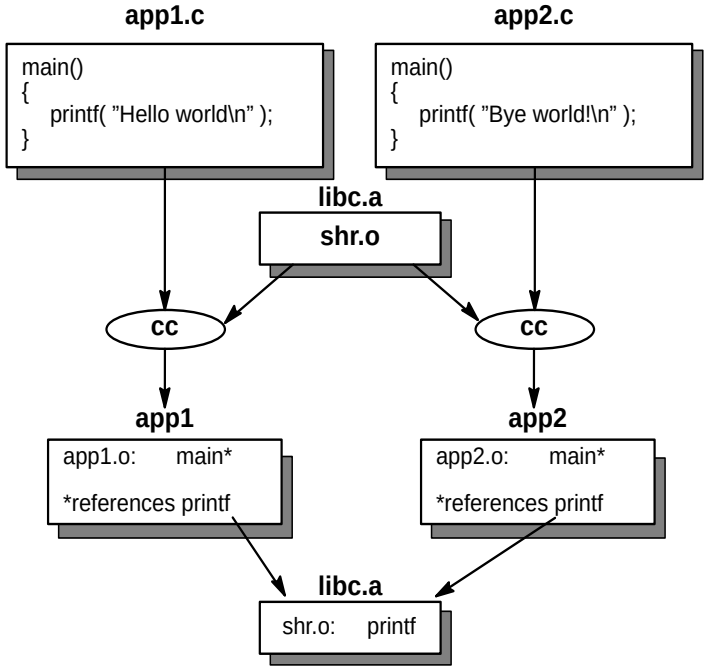
Relocating an Object Module

- ☞ The text section, or program code, is mapped to the text segment
- ☞ The runtime virtual address of the text and data segment origins are retained
- ☞ The applicable origin address is added to each of the relocation entries

Notes

Object modules and executable files are built to be address independent. That is, they are not bound to a particular runtime address. For this reason, object modules are often referred to as relocatable object modules. The linker uses the relocation information contained in the object modules to create an executable file. The relocation information specifies regions in the machine code that are memory references to either data or subroutines. The addresses of these references in the object module usually do not represent the runtime address of the data or subroutines. The loader uses the relocation information to patch the relevant locations in the machine code of the object module with the runtime addresses.

Code Organization for Shared Objects



Notes

Historically, linking multiple object modules was done statically. That is, the code contained in the text section of the various object modules was actually included in the final executable. Therefore, the executable was totally self–contained and required no further symbol resolution. This approach becomes inefficient when multiple executables reuse identical code, especially as this potentially reusable code grows in size. Disk space is wasted since copies of the same code appear in multiple programs. Changes to common code require relinking of every executable that uses this code.

AIX V3 allows for static linking (discussed later), but has enhanced facilities for dynamic linking. When dynamic linking is used, the text from the common modules is NOT included in the final executable. Instead, special entries are included in the XCOFF module to direct the system loader to the dynamically linked modules. At load time, the necessary object modules are loaded and any references to the text or data of the modules are resolved by the system loader.

AIX Systems Center

Advantages of Dynamic Linking

- ☞ Disk space is saved since routines may be shared among various applications
- ☞ An external, shared subroutine can be modified without relinking every application that uses that routine
- ☞ At execution time, only one copy of the shared routine exists in memory and is used by all applications referencing that routine
- ☞ Applications use memory more conservatively since they only need space for private data

Notes

A shared routine only resides in one location on the disk, not in the multitude of applications that utilize that routine.

If the internals of a shared routine are modified (for example, to fix a bug), the new routine can be made available immediately to every application; relinking an application is unnecessary.

Only enough paging space for one copy of the routine is required. This is more efficient utilization of system resources.

Each application using the routine must acquire space for the data requirements of that routine, but not the code.

Traditional Linker Technology

- ☞ Object modules and libraries must be organized according to references
- ☞ Library routines may only reference symbols in the same or subsequent libraries, or in explicitly specified object modules
- ☞ Errors resulting from multiply–defined symbols must be manually corrected by the programmer
- ☞ Archives must be processed by the **ranlib** command to avoid archive member ordering

Notes

Historically, linkers have depended on command line object module and archive ordering to resolve references to symbols contained in other object modules and in libraries. The linker builds a list of all external symbolic references and symbolic definitions for all input object files. This symbol table list represents the symbols that need resolution and those that provide definition. Symbol references are matched by name to symbol definitions, and, once resolved, the symbol table entry is discarded. The traditional linker processes library archive files differently than object files. Object files may have cross references, but library files cannot. This means that library files must be carefully placed on the linker command line. Library archive files are scanned only once, so any reference to a library–defined symbol must already be in the symbol table when the library is processed. The linker doesn't keep the external symbols defined in an archive library, so a referencing object module must precede the library on the linker command line.

AIX Linker Technology

- ☞ All symbols in all object modules and libraries are included by the linker
- ☞ Symbols may be defined before they are referenced
- ☞ When multiple definitions of a symbol occur, the first definition is retained and subsequent definitions are discarded
- ☞ The linker automatically utilizes shared objects

Notes

IBM's linker implementation has removed the need for command line ordering of object modules and archive libraries. The basic algorithm described before is followed, with the exception that library archive symbols are not discarded. The linker initially includes all symbols, discarding only symbol definitions that duplicate already defined symbols. The first definition of each symbol in the link always takes precedence and is used even if the first reference follows the definition. Traditional UNIX linker only retain symbols that are referenced as each object module is processed. Thus, a definition that is unused by the time an object module is fully processed is discarded.

When shared objects are referenced on the command line, the linker constructs the application to take advantage of those objects. The linker option **–bnso** can be used to statically link shared objects (physically) into the application. Static linking is an all–or–nothing prospect. For example, **–bnso** will also cause the system library routines that are used by the application to be linked into the application.

An Example of Shared Objects

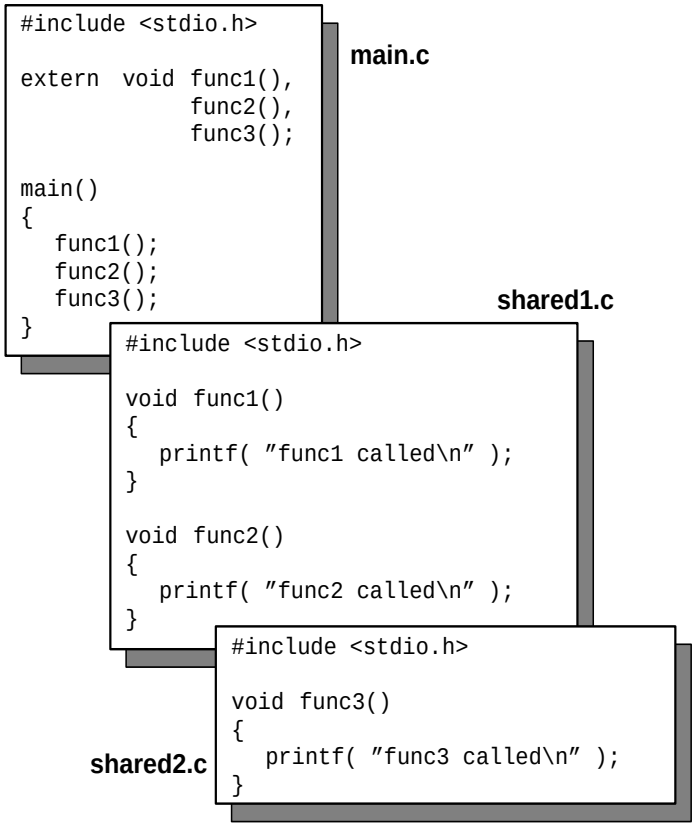
- ☞ Discuss program organization applicable to the use of shared object modules
- ☞ Demonstrate the use of the “Exports” file
- ☞ Introduce the steps required to build the shared objects, and to compile and link the application

Notes

This example is designed to introduce the student to the concept of shared objects. The use of shared objects requires some consideration of program design, as well as additional detail in the link process. This detail takes the form of a symbol export file which indicates to the linker what symbols are to be made publicly available to eternally referencing modules.

To accomplish this introduction, this exercise will step through the procedures necessary to design and build a small application that uses a shared object.

An Example: Program Source Code



Notes

This example will consist of:

- ✓ two source files to be compiled and linked to create a shared library
- ✓ the source file which references the shared modules
- ✓ the exports file necessary to explicitly export the symbols in the shared library that are referenced by other object modules
- ✓ the command lines to be used for compiling and linking the shared library and the module

AIX Systems Center

An Example: the Main Application vs. the Library Routines

- ☞ The main function represents that part of the code that is application–specific and non–sharable
- ☞ The routines **func1**, **func2**, and **func3** represent modules that are usable by other applications
- ☞ The shared objects may contain local data; while the code is shared by multiple copies of the application, each copy maintains its own private (or local) data

Notes

Consideration must be given to the organization of the application. The goal is to organize a programming project into (potentially) reusable code and application–specific code. The reusable code is targeted for a shared module, and specific functions from that code are then made available to the ”outside world” for use by any application.

An Example: the Exports File

- ☞ Lists the names of any routines and/or data structures that may be referenced by external modules
- ☞ The first line in the file names the linked shared object
- ☞ Each of the subsequent lines contain a single symbol

shrsub.exp

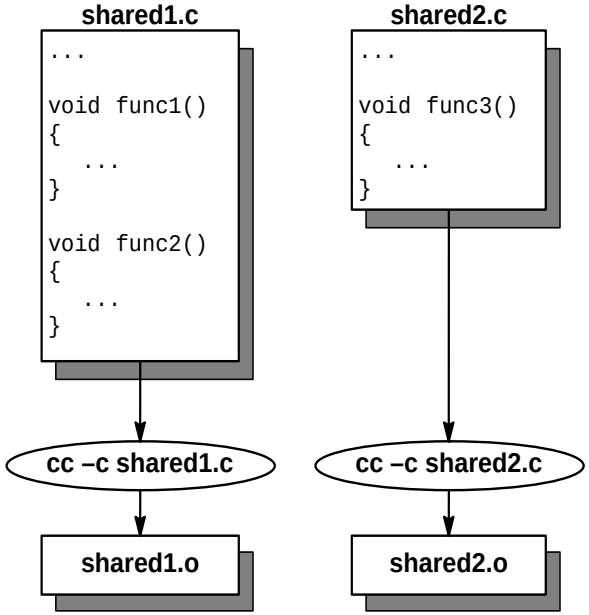
```
#! shrsub.o
func1
func2
func3
```


Notes

When building a shared library, it is necessary to provide a list of symbols that will be made available to referencing modules. This is done using an exports file. An exports file contains the symbols that are to be explicitly exported by a shared object module. The exports file also contains the name of the object module that defines these exported symbols. Modules that reference these exported symbols must import them at link time. This is accomplished by using the **–bI:***exportsfilename* flag, or by specifying the name of the shared object module on the command line. Resolution of these imported symbols is deferred until load time.

An Example: Creating the Shared Object

☞ Each source file is compiled to create separate standard object modules

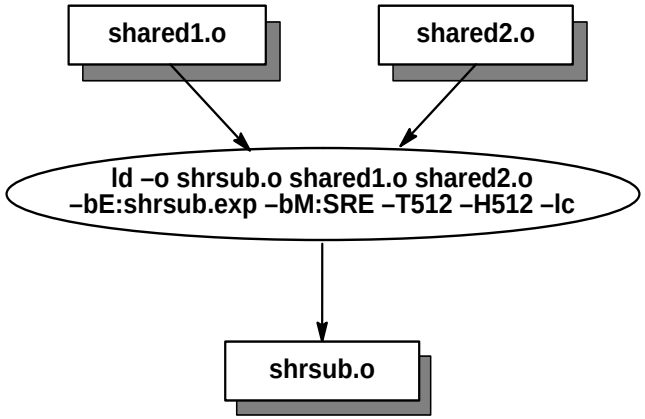


Notes

Standard techniques are used to compile the individual source modules.

An Example: Creating the Shared Object...

- ☞ The compiled object modules are combined to create a shared object
- ☞ Certain command line options are required for this step: **–T512**, **–H512**. These options remain consistent for this operation

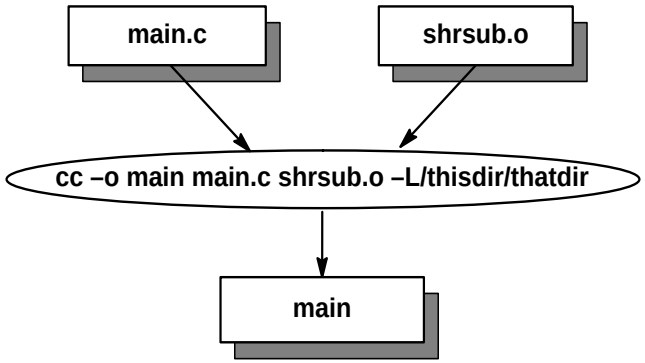


Notes

The **–H512** and **–T512** options specify alignment information. This is a magic incantation for the linker and specifies alignment information for the text and data segments. Looking at the contents of both the **/etc/xlc.cfg** and **/etc/xlf.cfg** configuration files, you can see that the stanzas within both specify these options for the linker.

An Example: Linking the Application to the Object

- ☞ The non–shared portions of the application are compiled in the normal manner
- ☞ The shared object is specified on the command line, but the linked application only *references* the shared code that resides within the object
- ☞ The directory named using the **–L** option is stored within **main**



Notes

The program **main** is now executable. The symbols **func1**, **func2**, and **func3** have been marked for deferred resolution. At runtime, the system loader will place the shared object in the shared text memory segment (unless it is already loaded) and dynamically resolve the references. To find the object modules, the library search paths are used. The default search paths are the **/lib** and **/usr/lib** directories; these paths are built into the linker but can be modified or replaced (details later). For this example, the working directory is **/thisdir/thatdir**, and this path is added to the search paths with the **–L** option.

Exactly how does a symbol get marked for deferred resolution? This is discussed on the next foil.

If several shared object modules are to be used by an application, it is possible to group the modules into an archive library. Compile and link each of the shared object modules (using the steps shown), and use the following command to create the archive library:

```
ar rv libshr.a shrsub.o anothershr.o yetanothershr.o
```

The command line to compile and link an (arbitrary) application using this archive library would be:

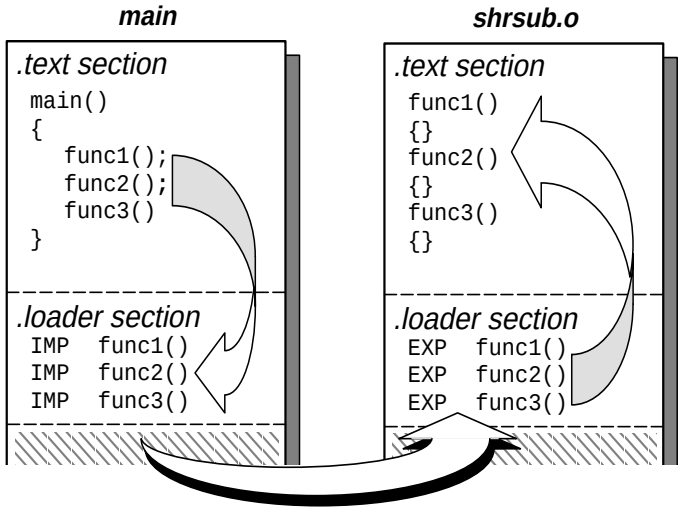
```
cc –o mainapp mainsrc.c –L/u/sharedlib –lshr
```

Of course, the name of the archive library and the names of the shared library object files vary according to the application.

An Example: Resolving Symbols at Runtime

- The main application references the symbols **func1**, **func2**, and **func3**. The code for those symbols continues to exist only in **shrsub.o**

- When the application is loaded into memory, the code in **shrsub.o** is also loaded. Recall that **main** contains the information necessary to find **shrsub.o**



Notes

The actual code for **func1**, **func2**, and **func3** is not physically copied to the main application file.

During the final link (illustrated on the previous foil), the linker finds the references to the symbols **func1**, **func2**, and **func3**. While processing the shared object, the same symbols are found as exported from that object. The linker then marks those symbols as imported to the main application from the shared object. The name of the shared object is stored in the main application, along with the current library search paths. The application object module is restructured and global linkage code is added at every reference to an external function; this code manages the details of jumping to an external object, executing code, and returning to the original module to resume execution. It is this global link that provides the performance penalty in using shared objects. During execution,12 machine cycles are required to complete each and every jump to an external routine and begin execution in that routine.

Note that only the filename of the module is stored in the referencing module; the path to the module must be explicitly specified. This is accomplished with the command line option **–L**<*path*> , discussed in detail later.

Before the application begins execution, the system loader searches for any symbols marked for deferred resolution. For each of these symbols, the object module containing that symbol is loaded, if necessary. Housekeeping concerning addressing from the main application to the shared object is completed, and execution begins.

AIX Systems Center

Linker Flags: –l<libname>

- ☞ Specifies a particular (archive) library to be searched
- ☞ The name supplied is used to construct the name of a library file on disk. This name is prefixed with “lib”, and a “.a” suffix is added. For example, the math library is specified on the command line as **–lm**; the actual name of the math library file is **libm.a**
- ☞ The option is a lowercase “L”
- ☞ The compiler configuration files specify a set of required libraries for the associated compiler

Notes

This is a (de–facto) standard command line argument used to name libraries or archives.

Linker Flags: –L<path>

- ☞ Lists additional directories to be searched by the linker
- ☞ Adds directories (to the built–in library paths) that are to be searched by the system loader
- ☞ When specifying multiple paths, the **–L** option is used for each separate path
- ☞ If path(s) are listed using this option, the **LIBPATH** environment variable is ignored
- ☞ To permanently add a particular path for your applications, set (or modify) the **LIBPATH** variable

Notes

These built–in paths are in the linker source code. Recall that the library search paths are stored in the executable program, and are used by the loader when locating shared objects.

When libraries are specified using the **–l** option, additional directories to be searched during the link can be specified with the **–L** option. All paths specified in this manner are searched before the built–in search paths. Note that the first occurrence of a library is the one that is used.

An environment variable, **LIBPATH**, may contain a list of library search paths. This variable is mutually exclusive with the use of **–L** and the built–in paths.

The hierarchy of command line options and environment variable is:

- ✓ If **–L** is specified at least once, the listed paths (along with the built–in paths) become the library search paths. This **LIBPATH** environment variable is ignored.
- ✓ If the **LIBPATH** variable exists, and **–L** is not used, only the contents of the environment variable are used to specify the library search path. The built–in paths are not used.
- ✓ If **–L** is *not* used and the **LIBPATH** variable is *not* set, only the built–in paths are used.

To modify the **LIBPATH** variable, add a line similar to the following to the **$HOME/.profile** file:

```
LIBPATH=/u/me/mylibdir:/lib:/usr/lib
export LIBPATH
```

LIBPATH at Runtime

- ☞ Used to change the search locations for libraries
- ☞ Contents are used before those specified when the object or application was linked
- ☞ Due to the mechanism for examining the list of search paths, the current working directory is also searched

Notes

The environment variable **LIBPATH** be used to list a set of directories that should be scanned to locate shared objects. The syntax for the contents of **LIBPATH** are exactly the same as illustrated earlier:

```
export LIBPATH=$HOME/lib:$HOME/test/mylib
```

This will cause the loader to search for objects of the appropriate name in the **lib** and **test/mylib** directories under the user's home account. The object names must, of course, match the names for which the loader is searching. As an illustration, consider the example of building a shared object given earlier in this module. Once the shared object **shrsub.o** is archived in **libshr.a**, it would be moved to a "standard" location, **$HOME/lib**. The user could then link an application to **libshr.o** using the command:

```
cc –o mainapp mainsrc.c –L$HOME/lib –lshr
```

Consider the necessity of modifying one of the routines within the shared object, and using the existing application **mainapp** to test it. The new object is built in the directory **$HOME/test/mylib** and is archived into **$HOME/test/mylib/libshr.a**. To use the existing copy of **mainapp** to test it, the user only needs to set the envinronment variable

```
export LIBPATH=$HOME/test/mylib
```

and run the application. The new shared object in the **test/mylib** subdirectory will be loaded and executed. The search of this subdirectory will occur first, and the new version of **libshr.a** will be found first.

Linker Flags: –bE:<exports file>

- The exports file contains a list of symbols
- It is used to name routines that are to be visible to referencing modules. The named symbols are marked as *exported* in the .loader section of the defining module
- The same file can be used as an imports file in lieu of naming a library during linking. Note, however, as an imports file, it must contain a reference to the defining object module (using the **#!** notation on the first line of the file). Including this line does not affect its use as an exports file

Notes

Linker Flags: –bM:<module type>

- ☞ The default module type is **1L**. This specifies a single–use object with private data
- ☞ **S** indicates that the module is shared, and should be loaded into the shared library segment
- ☞ **RE** specifies a module as reusable. It requires a private copy of data for every process that uses the module. This is generally used (along with the **S** module type) to create a shared library

Notes

An additional option, **–SRO**, is designed to specify shared, read–only code. At this time this is equivalent to **–SRE**.

Linker Flags: –bnso

- Used to statically link object modules that have been defined as shared
- When this flag is used, a private copy of code from the shared objects is included in the final executable
- When this option is used with system libraries, the option **–bI:/lib/syscalls.exp** must be used when linking

Notes

The file **/lib/syscalls.exp** must explicitly be named as an import file when statically binding to system libraries; e.g. **libc.a**.

Since the shared object is physically bound to the application, the resulting file will be larger than if the shared libraries were used. Sometimes this can be a significant difference in size.

Linker Flags: –bloadmap:<filename>

- ☞ Indicates that the linker should produce a file containing detailed information about the link process
- ☞ Link errors and messages are stored in this file, along with the internal binder commands used during the link
- ☞ The information contained in this file can be beneficial for debugging purposes

Notes

The loadmap will contain, for example, a list of multiple declarations of symbols, where those multiple declarations were found, a list of object modules and archives that were processed to build the executable, and the phases that the binder goes through in during the build process.

Control Sections, or “csects”

- ☞ The XL family of compilers generate object modules that contain one or more *csects*
- ☞ A csect contains executable code or data
- ☞ The contents of a csect maintain a constant relationship; the csect is viewed as a self–contained unit by the linker
- ☞ csects containing code are collected by the linker to form the .text section of an executable; those containing data are collected to form the .data section

Notes

IBM compilers generate object modules that are composed of one or more control sections (csects). A csect is a unit of executable code and/or data generated by a compiler or assembler for which all elements retain a constant relationship to one another. A csect is the smallest separately relocatable unit of a program. The linker collects csects from multiple object modules into either a read only area (text) or a read write area (data) to create an executable.

AIX Systems Center

Linking and Binding

Duplicate Symbol Definitions

- During a link, the first occurrence of a symbol takes precedence, even if the first reference follows the first definition
- If a duplicate symbol is encountered, it is discarded, along with the csect containing that duplicate symbol. Since the discarded csect may contain other symbols as well, these other symbols are also discarded. As a result, unresolved symbol errors may occur at the end of the linking process
- The command line option **–bnodelcsect** can be used to inform the linker that only the duplicate symbol itself should be discarded, not the entire csect
- The linker has the liberty to reorder csects to place symbol references near symbol definitions. The goal is to reduce paging activity in the executing application
- The compilers can generate a csect containing a single address constant. These csects are grouped together to form the TOC

Notes

As mentioned, the first definition of a symbol in a link takes precedence and is used even if the first reference follows the definition. When a duplicate symbol is encountered, the default action is to discard this symbol along with the csect that contains it. All symbols contained in that csect are therefore discarded.

This approach may result in undefined symbol errors if these symbols are referenced. This default behavior can be changed using the **–b** flag with the suboption **nodelcsect**. This flag causes only the duplicate symbols to be discarded, not the entire csect, thus eliminating any undefined symbol errors caused by csect deletion.

APAR IX16036 is available to make **nodelcsect** the default option under AIX version 3.1. This option is the default under AIX version 3.2.

An additional feature of the IBM linker is csect reordering. Code csects are sorted based on the use of symbols by other csects; this is done to reduce paging by placing definitions close to uses. An addtional option for the binder, **noobjreorder**, bypasses the csect reordering phase. This option is included in the new version mentioned above.

Table of Contents (TOC)

- ☞ The linker collects all csects of storage class TC and builds a contiguous area called the Table Of Contents
- ☞ The TOC is stored at the end of the .data section
- ☞ Function descriptors are stored immediately before address constants
- ☞ Address constants within the TOC are indirect references to the actual data storage locations
- ☞ Most load time relocation activity is concentrated in the TOC

Notes

A compiler or assembler can generate csects each containing a single address constant. The link process will collect all the csects with a storage mapping class of TC into a common area called the Table of Contents, or TOC. The TOC is a collection of all the address constants of the program into a contiguous part of the program data area. All Table Of Contents entries are placed at the end of the data section, preceded by any function descriptors. The address constants contained in the Table Of Contents are indirect references to the actual data. This concentrates most load time relocation activity into a small part of the data section.

The maximum number of symbols in the TOC is 16,000.

Dynamic Linking

- ☞ Combined with the .loader section, the TOC is used to resolve symbols at load time
- ☞ Information about the location of the symbol definition is included in the .loader section of the executable file
- ☞ A TOC entry is created for each imported symbol. It is initialized to 0 (zero)
- ☞ At load time, the object module defining the referenced symbols is loaded. TOC entries for imported symbols are filled in with the actual runtime address
- ☞ The search path is used to locate the named object modules and libraries

Notes

Dynamic linking in AIX V3 is accomplished using the XCOFF loader section and the Table Of Contents. The loader section contains information used by the system loader to dynamically resolve symbol references to external symbols. These symbols are marked as imported during the link, and information concerning the location of the actual symbol definition is included so the loader knows where to go to resolve references. Each imported symbol in the loader symbol table carries with it the name of the object module that defines the symbol, and special information concerning the symbol type and its storage class. Also included in the loader section is the search path used to locate the defining object modules.

Function Descriptors

- When a symbol referring to a function is exported from an object module (using the **–bE:**<*file*> option), a function descriptor is created in that module's TOC
- The TOC entry for a symbol referring to an external function points to a data structure called a function descriptor
- Global linkage code (**glink.o**) is added to the .text section for every imported symbol that references an external function. This code uses the information within a function descriptor to manage the flow of execution of a program

Notes

If the imported symbol is a function symbol, the TOC entry points to a function descriptor located in the TOC of the module that defines the function. At link–time, global linkage code is added for each imported function symbol. This global linkage code accesses the function's descriptor indirectly through the TOC entry in the original module. The descriptor contains the entry point address for the function code, and the address of the TOC for the defining module. These values are used by the global linkage code to properly transfer control to the function code.

Function Descriptors...

- ☞ After relocation is completed, each function descriptor contains the entry point address of the target function
- ☞ The address of the defining module's TOC is also contained in each descriptor
- ☞ Global linkage code manages the task of saving the current TOC address, loads the TOC address of the defining module (from the function descriptor), and branches to the entry point of the function

Notes

Function descriptors are created in the TOC of the defining module if the function symbol is explicitly exported using the **–bE:**<*exportsfilename*> flag on the link command line. The function descriptor is 12 bytes long. The first word (4 bytes) is the address of the entry point for the function code. The second word is the address of the TOC in the defining module. The third word is an environment pointer for languages like Pascal and PL/I. For C and FORTRAN, this third word is filled with zeros and is not used.

Relinking an Application

- An executable file retains all the information necessary to relink that file
- To replace an individual routine, it is only required to relink the modified object module with the executable file
- Relinkability can reduce the time required to apply changes to an application
- Due to the precedence rules for symbols, the new object module(s) must be specified before the executable on the command line

Notes

An important feature of AIX V3 that is often overlooked is the relinkability of executable modules. In other implementations, if a bug is found in one of the modules that was included in **a.out**, it is necessary to change that module, compile it, and then relink all of the component object modules that make up a.out. In AIX V3, **a.out** can be used as input to a link. This means that if a bug is found in one of the component object modules, that module must be changed and recompiled, but it is not necessary to relink all of the component object modules. AIX V3 allows the changed module to be linked directly with a.out

Relinking an Application...

- ☞ Assume that the executable file is named **a.out**, and contains routines from **file1.c**, **file2.c**, **file3.c**, **file4.c** and **file5.c**
- ☞ Suppose a bug is discovered in **file2.c**, and the code is modified. After compiling **file2.c** to create **file2.o**...
- ☞ The link command necessary to rebuild the application with the new module is:

  `cc file2.o a.out`

- ☞ A new a.out file is created which contains the modified **file2.c** code, as well as the original code from files **file1.c**, **file3.c**, **file4.c** and **file5.c**

Notes

For example, assume **a.out** consists of **file1.c**, **file2.c** and **file3.c**, **file4.c**, and **file5.c**. If a bug is found in **file2.c**, then **file2.c** must be corrected and recompiled. To produce an updated executable, only **file2.o** needs to be linked with **a.out** using the command

```
cc file2.o a.out
```

Relinkability, used correctly, can reduce the time required to link an application. Applications should be split into several modules to take full advantage of relinkable executables.

Linker Performance

- ☞ The use of third–party archive libraries usually produces performance degradation. This can be due to a large number of symbols, all of which are loaded, and some of which require resolution
- ☞ AIX system libraries are shipped in binder–output format. Thus, references between symbols within a library are already resolved. This results in less work for the linker

Notes

The AIX V3 binder (linker) has been criticized as being slower than the linkers of competitive systems. A substantial part of binder execution time is due to the retention of symbols required to build a relinkable executable. If relinkability is not exploited, then large applications will indeed take longer to link. Linking to large archive libraries causes a definite performance degradation. The system subroutine libraries such as **libc.a** are shipped in binder–output format, as opposed to an archive of .o files. References within the library are already resolved during the system build process, so only references from a user application to the library need to be resolved.

Prelinking a Library or Archive

- ☞ System libraries contain a large amount of shared code. This implies that much of the symbol resolution is deferred until load time

- ☞ Third party libraries can be easily prelinked to reduce symbol resolution overhead. Given an archive with the name **libparty.a**, the following command creates a module in binder–output format:

  `ld –r libparty.a –o libpartya.o`

- ☞ The **–r** option disables garbage collection

Notes

However, many third–party subroutine libraries are shipped in archive form as raw .o files. This introduces substantial overhead into the link process since the binder has to do symbol resolution for the entire library each time the application is bound. The performance difference between bound and unbound libraries shows dramatic improvement. To pre–bind a third party library, use the **ld** command with the **–r** option as in the following example.

```
ld –r libparty.a –o libpartya.o
```

This prebinds **libparty.a** to resolve any internal references and produces the prebound library **libpartya.o**. The object module **libpartya.o** is then used on a command line to resolve references from a user application to the library.

Note that this object module will be statically bound to the application.

Garbage collection is the process of deleting any unused or unreferenced symbols.

Creating a Shared Object From a Library or Archive

☞ First create an exports list for the shared object

```
/usr/ucb/nm archive.a | grep ' [BAD] ' \
   | cut -f3 -d' ' > archive.exp
```

☞ Then create the shared object

```
ld -o shr_arch.o archive.a -bM:SRE \
   -bE:archive.exp -H512 -T512 \
   <libs> -lc
```

☞ When linking to the shared object, indicate the location of the shared object for load time resolution

```
xlc <objects> shr_arch.o \
   -L/archive/path <other options>
```


Notes

To turn an archive file into a shared object, first generate an exports list for the symbols using the Berkeley version of **nm**. This version of **nm** more clearly indicates which symbols are actually defined within the archive. Note that the command above is a single command. The grep search pattern should have a space both before and after the capital D. The **cut** field delimiter is a space (the **–d** option). The export list file should be named appropriately.

The command to create the object is shown. An output filename is specified as well as the export list generated in the previous step. The **–bM** option indicates that the code should be marked shared–reentrant. The **–H512** and **–T512** are alignment options (see the **/etc/xlc.cfg** file contents for the default options for compiling and linking). Any libraries required to resolve symbol references should be included as <*libs*>, as well as the standard C library.

Note also the addition of the **–lc** argument. Any libraries containing functions that are used by anything within **archive.a** must be specified on the command line to complete the symbol resolution process. This implies that much symbol resolution is done when this object is created, not when the final application is built.

This Page Intentionally Left Blank

Notes

The new shared object should be moved to **/lib** or **/usr/lib**. These are the standard search directories. The file may also be renamed to resemble an archive name; **libshr_arch.a** for example. (The linker understands the contents of the file via the magic number; the name is merely a formality.) This would allow the option **–lshr_arch** to be used on the command line:

`xlc <`*`objects`*`> –lshr_arch` *`etc...`*

If a standard library search path is not used, however, the **–L** option can be used to add an additional path to the search list; an appropriate site–specific location might be **/usr/local/lib**.

If the object is to remain in the development working directory, the command line option might be **–L$PWD** (assuming the Korn or Bourne shell).

Runtime Loading and Binding

- ☞ The **load()** system call can be used to install an object module into an executing program. The system loader is used to bring the module into memory
- ☞ Symbol resolution can take place when the object is loaded, or at a particular point in the program. This later resolution is accomplished with **loadbind()**
- ☞ The **unload()** routine can be used to remove object modules from an executing program. These modules must be loaded with the **load()** routine

Notes

In addition to link time and load time binding, AIX V3 offers system calls to facilitate runtime loading and resolution.

The functionality of the loader is directed towards a single executable. The **load**() system subroutine encapsulates that functionality, allowing a running program to invoke the loader, access object modules, and expand its own capabilities.

A large application can be split up into two or more object files in one of two ways that allows execution within the same process. The first way is to create each of the application's object files separately and use **load**() to explicitly load an object when it is needed. The other way is to specify the relationship between the object files when they are created by defining imported and exported symbols.

Object files can import symbols from other object files. Whenever symbols are imported from one or more other object files, these object files are automatically loaded to resolve the symbol references if the required object files are not already loaded, and if the imported symbols are not specified as "deferred resolution" (the L_NOAUTODEFER flag with the **load**() system call).

The **load**() subroutine loads the object file for the program into the calling process. Unlike the **exec**() subroutine, **load**() does not replace the current program with the new one. Instead, it loads the new program into the process private segment at the current break value; the break value is updated to point past the new program. Upon successful completion, the **load**() subroutine returns the pointer to the function that is the main entry point of the module.

This Page Intentionally Left Blank

The **unload**() subroutine is used to unload an object module that was added using **load**(). When the calling process uses the **unload**() subroutine to unload the object file, the space is unusable by the process except through another call to the **load**() subroutine. If the kernel finds an unused space created by a previous unload, rather than load the program at the break value, it loads the program into this unused space. Space for loaded programs is managed by the kernel, not by user–callable storage management routines.

Using the **load**(), **unload**(), and **nlist**() subroutines, it is possible to dynamically link unrelated object modules. The **nlist**() subroutine allows a program to examine the name list in the named executable file. It selectively extracts a list of values and places them in the array of nlist structures passed as a parameter. The symbol values returned are actually relocatable addresses that are used by the loader at runtime for dynamic relocation. Therefore, it is necessary to manipulate certain addresses in order to obtain function addresses in unrelated object modules.

System Limitations

- ☞ Program text is limited to a single segment of 256MB
- ☞ The shared text segment is limited to 256MB
- ☞ The process private data segment is limited to 256MB
- ☞ 10 shared data segments are available simultaneously from a single process
- ☞ A shared data segment is limited to 256MB
- ☞ Adequate paging space must be available

Notes

Using a combination of regular and shared text, a potential text size of 512 MB may be reached. Using dynamic linking techniques, another (approximately) 240 MB of text space is available (recall that the **load()** function places a loaded object into the data segment).

The maximum amount of private data per process is 256MB. This limit is lowered via the *data* value in the file **/etc/security/limits**. Individual users may have lower limits in their individual stanzas. The value in this file is in blocks; therefore, 256MB is equivalent to 524288 blocks. If a change is made, the user must logout and login for the new limitation to take effect.

If an application requires a large amount of text or data space, the paging space on the system must be large enough to contain the application. Use the **size** command to determine the application's requirements. If either the data limit or paging space is too small, the user will receive an "exec error" when attempting to execute the program.

Linking Very Large Programs

☞ If the error "0706–547 RLD in <*filename*> is not in a valid section" occurs, the program space during the bind has exceeded 2^{32} bytes.

- ❏ To spread out linking overhead, use incremental links to create partial object modules

☞ If the error

```
Could not load program
Error was:
```

occurs, then the text or data section has exceeded 256MB

- ❏ Utilize shared objects to move a portion of the code to the shared text segment
- ❏ Utilize shared memory to store a portion of the data

Notes

If a program is very large, its requirements may exceed the system limitations on the maximum text or data segment size. Alternative programming techniques may be required to sidestep these limitations.

To effectively use incremental linking, organize the code into logical groups. Link each logical group into an intermediate object module using the **–r** option. This allows the linker to resolve symbol references internal to that module. The final link of the application will see a performance improvement as a result.

Configuration Tips

- ☞ Linker performance bottlenecks include
  - ❑ Inadequate paging space
  - ❑ Inadequate physical memory
  - ❑ User limits set too low
- ☞ Potential error messages:
  - ❑ “Out of Resources”
  - ❑ “Killed by signal 9”
  - ❑ “Page space low”

Notes

The linker requires a generous amount of virtual memory; at least twice the size of physical memory, if not three times the size, is recommended. If there is not a reasonable amount of physical memory available, excessive paging will occur. Determining an acceptable system configuration is dependant upon the type of development that will take place.

The user limits should be set to realistic values. In the **/etc/security/limits** file, the following values are recommended:

```
stack = 65535
data = 262144 (or 524288)
```

If the compiler emits an “Out of Resources” error message, ensure that the **/tmp** filesystem is of adequate size. Also, inadequate paging space may generate a “Killed” or “Page space low” message.

To enforce a single compile and link at a time on the system, utilize the queueing system and the **enq** command.

Useful Commands

- **nm**
  - **/usr/ucb/nm** is the BSD version of this command
- **dump**
- **ar**
- **ranlib**
- **strip**
- **size**
- **slibclean**

Notes

The commands **nm** (**/usr/usg/nm** by default) and **/usr/ucb/nm** produce similar information, but in different output formats. They both write the symbol table of the named object module(s) to stdout.

The **dump** command writes selected parts of an object module to stdout. Parameters cover every section of an XCOFF module.

The archive program **ar** combines one or more files into a file in archive format. This command is most often used to create object module libraries.

ranlib is used to convert archive libraries into random libraries. A symbol table containing every symbol in the library is built and inserted into the library.

strip reduces the size of an XCOFF module by removing the linking and debugging information. Note that executables that are stripped are no longer relinkable.

To determine the resource allocation of an application, the **size** command reports on the requirements of the individual sections of an object module or executable image.

Once a shared object becomes unused, it will remain in memory for an arbitrary period of time. The **slibclean** command flushes unused objects from the shared text segment. This is useful when repeatedly building and testing a new shared object.