AIX linking 102

Skill Level: Intermediate

Gary Hook (ghook@us.ibm.com)
Senior Technical Consultant
IBM

03 Apr 2002

Are you writing or porting applications to AIX? Get a quick introduction to the most commonly used features of the linker and loader, plus practical tips and techniques. This short tutorial from AIX expert Gary Hook focuses primarily on the library search path.

Section 1. Tutorial roadmap

What is this tutorial about?

This tutorial is for developers writing for or porting applications to AIX and who want a quick introduction to the most commonly used features related to the linker and loader. The second in a series of practical tips and techniques, this article focuses on AIX runtime linking and the slibclean command.

Should I take this tutorial?

Before you begin, you should have a working knowledge of application architecture, including shared modules (or shared libraries). You should also understand the role played by the linker in application construction.

After completing this tutorial, you will understand:

- How best to use certain linker options related to runtime linking
- Why your runtime linking-enabled application may not produce the results you expect
- What work is done by the kernel loader for a runtime linking application

- Effective use of the slibclean command

Tools needed

The runtime linking feature discussed in this tutorial was initially made available in AIX 4.2. Later levels of the operating system provide behavior compatibility, including support for 64-bit programs in both AIX 4.3 and 5L.

The techniques in this tutorial apply to the linker, loader, and shell; unless stated otherwise, they are independent of the application development language. Therefore, applications written in C, C++, Fortran, Cobol, etc, can all be affected in a consistent manner. No specific language or version of compiler is required, nor is any level of AIX, other than a minimum of 4.2.

Section 2. AIX Introduction

Introduction

This is a continuation of a series started in the Linking 101 tutorial. With this tutorial, you will gain an understanding of the nuances of runtime linking and the technical reasons why things should be done in certain ways. You will also pick up some pointers that will assist in deciding if runtime linking is right for you. The tutorial concludes with a brief discussion of the slibclean command.

As I stated in the introduction of Linking 101, my goal here is to pull together insights and advice based on my experience, and to offer suggestions in a concise form. The opinions herein will not necessarily fit every situation. Even so, they will increase your understanding of the nuances of these particular features and assist you in determining how these AIX functions apply to your situation.

Section 3. Runtime linking

The truth about -brtl and -G

In the years since AIX 4.2 was introduced and runtime linking was added to the feature set of the linker and loader, there has been no set of options more misunderstood. The first and most fundamental point (that is, the rule) about these options is:

```
-G and -brtl are not equivalent
```

The "`-G`" option is designed to make the task of building shared modules simpler, while at the same time adding a feature that provided a (much-requested) behavior change. The "`-brtl`" option is designed to be the quick option that enabled an application for runtime linking.

They serve different purposes and have never intended to be interchanged. In fact, by carefully reading the AIX documentation on the ld command, you will find that `-G` is actually a pseudo-option". It's shorthand notation for the options "`-brtl -bnortllib -bnosymbolic -berok -bM:SRE`". The most common problem I run into is situations where it's been presumed that the option pair "`-brtl -bnortllib`" is just as good as (read: functionally equivalent to) "`-G`". And if it were as simple as those two options, then sure, they would be equivalent.

By reading the documentation very carefully, you discover that those two options aren't the issue. The real issue is the "`-bnosymbolic`" option. This option accomplishes a number of useful tasks which are not normally done by the linker when building a module. Symbols get added to the loader symbol table, which is used by the runtime linker for symbol resolution, and these symbols can be rebound at runtime. The default option, "`-bnosymbolic-`", disallows rebinding of symbols when both the reference and the definition (i.e. target) are in the same module. (Note the two different forms of this option; the trailing minus sign shown in the default form is intentional. Refer to the AIX documentation for more details.) Let's take a look at a simple example that illustrates this fundamental problem.

Misusing the -brtl option: an example program

My test program consists of a simple main routine that calls a function in a shared module. That function then calls another function that is defined within the same module. This latter function is intended to be replaced by an alternative definition supplied by the main app, or by another module, via runtime linking. Starting with the shared module, consider the top-level routine foo() from file `foo.c`:

Listing 1. Top-level routine foo() from file foo.c

```
#include <stdio.h>

extern  void    bar( void );

void foo( void )
{
    printf( "In foo(), %s, calling bar()...\n", __FILE__ );
    bar();
}
```

This function calls out to bar(), defined like this in file `bar.c`:

Listing 2. An example

```
#include <stdio.h>

void bar( void )
{
    printf( "In bar(), %s, calling bar()...\n", __FILE__ );
}
```

Both of these files will be used to build libfoo.so, and both foo and bar will be exported from the module. Next, our main function looks like this:

Listing 3. libfoo.so

```
#include <stdio.h>

extern  void    foo( void );
extern  void    bar( void );

int main( int argc, char *argv[] )
{
    printf( "In main() calling foo()...\n" );
    foo(); /* in the shared module */
    return( 0 );
}

void bar( void )
{
    printf( "In new bar() provided by main app...\n" );
}
```

We supply an alternative definition of bar() which is intended to be used by any routine needing to call a function with that name. The main routine calls foo in the shared module, which in turn should call bar back in the main application.

N.B. You may notice that I've chosen to separate the functions foo() and bar() in separate compilation units. This is intentional. If you choose to try out this example, and place both functions in the same source file, be sure to add the `-qfuncsect` compiler option to ensure that the reference to bar() can be intercepted by the linker. For more information, see the compiler documentation and the Linking and Loading Mechanisms document listed in the References section.

Misusing the -brtl option: building and testing

Here's the makefile:

Listing 4. The makefile

```
CFLAGS=         -g
LDSOFLAGS=    -brtl -bnortllib -bM:SRE -bnoentry

all:    libfoo.so main

main:   main.o
        xlc -brtl -o $@ main.o -L. -lfoo -bexpall

libfoo.so:      foo.o bar.o
        ld $(LDSOFLAGS) -o $@ foo.o bar.o -lc -bexpall
```

The important thing to notice is the set of options used to build the shared module. The LDSOFLAGS variable contains an incorrect, although commonly used, set of options to build runtime-linking-enabled modules. The main application is built with the required `-brtl` option, plus a library search path directive. Both the app and module use the linker's `-bexpall` option to export symbols, avoiding the need for an export list.

Build ('make') and run the program. (You can ignore the linker warnings about duplicate symbols; we already know there's more than one definition of bar().) You'll see the following output:

Listing 5. Output

```
$ ./main
In main() calling foo()...
In foo(), foo.c, calling bar()...
In bar(), bar.c, calling bar()...
```

One of the primary reasons for using runtime linking is to allow symbol preemption to occur based upon command line ordering. In this example the main app is first; libfoo.so is second, followed by any required system libraries that compiler may silently reference. The intent is that the definition of bar() provided by the main application is used instead of the one provided by libfoo.so. But you can see that the output doesn't match our goal.

The problem here is not the organization of the program, but rather the way that libfoo.so was built. As stated earlier in this section, `-G` and `-brtl` are not equivalent. The former is designed for building shared modules; the latter for main applications. We did that last part for main, but we must change the way the shared module is built. Change the makefile so that ld flags look like this:

```
LDSOFLAGS=      -G -bnoentry
```

Delete libfoo.so and make it again. Run the program:

```
$ ./main
In main() calling foo()...
In foo(), foo.c, calling bar()...
In new bar() provided by main app...
```

You can see that the desired results have been achieved. The new definition for bar(), provided by the main executable (i.e. the first module in the process), is used by the shared module.

You can see that this one detail can make the difference between an application that works as intended, or behaves oddly or inconsistently. To build shared modules that provide the same symbol preemption capabilities as you might find on other Unix systems, be sure to use `-G` when building modules and `-brtl` when building main applications.

Runtime Linking at runtime

On AIX, the historic behavior of applications and modules is that symbols are resolved at link time (when the module or executable is built). Runtime linking changes all that, allowing the developer to build modules that have their symbols resolved when the module is put into use, and allowing applications to contain multiple definitions of symbols that are all nicely resolved at runtime to a single instance, which all parts of the application agree to use.

The subtlety here is that runtime linking is an attribute of an application, not a module. It is the application that defines runtime behavior (in this context, at least), and you the developer create your executable to implement your application design. A module, on the other hand, can be built to cooperate with runtime linking or not. Again, you the developer design and build according to your desired behavior. More on the internal structure details in a bit.

What, exactly, does runtime linking on AIX do? Let's see:

1. It uses the loader symbol table of the executable and dependent modules as the APIs for those modules.
2. It detects multiple instances of a symbol and directs all references to that symbol to a single instance.
3. It causes the loader to keep track of module load order based on the command line used to build the modules.
4. At exec or load time, it detects unresolved references to symbols and dynamically finds a definition for the symbol in all of the modules in the process.
5. Locates initialized and uninitialized instances of a data structure (in different modules) and redirects all references to the first initialized instance (or, if none exist, the first uninitialized instance).

You can see that this behavior allows you to build modules that are not fully resolved, which is contrary to the traditional AIX methodology of link-time symbol resolution. Complex applications, where the modules are inter-dependent, can be challenging to build when link-time resolution is required. By deferring resolution to runtime, your application components can be constructed with the assurance that symbol definitions will be available when needed.

You also have the option to construct modules that each contain local definitions of symbols, then let the runtime linker decide which instance of a symbol to use (like the example program above). This can be especially convenient when dealing with global data structures. And for the reader with experience on other Unix systems, it is often convenient to use command line ordering to dictate which symbol should be used by all references in a process.

Runtime symbol resolution

You've read about the ability of the runtime linker to resolve symbol references at runtime. What about during execution of the program?

On AIX this runtime fix-up occurs either at exec time, before main is entered, or at the time a module is dynamically loaded. Fix-up does not occur when a symbol is used. This is a distinct difference compared to other systems. It means that the work is done at one time, and it means that any outstanding references in a module will have to be resolved when that module is loaded (e.g. via the dlopen() function). If your running program can not satisfy the requirements for every symbol needed by the module you are loading, the load will fail. Approximately 90% of the time this is the reason for loading failures in RTL programs. (The other 10% is usually due to dependent modules not being found in the library search path.)

If you've read other linking documents (see References at the end) you understand that AIX promotes a "black box" paradigm for modules. That is, a module has a well-defined API and from the outside behaves in a defined way. The internals of the module, and the use of any dependents, should be considered "opaque" to the using module(s). Therefore, as you construct your RTL application, consider the source of necessary symbols. If you require an arbitrary symbol to be used by a dynamically loaded module, where will that symbol come from? Will another module that supplies that symbol be already loaded? Will the main application provide that symbol? If you start making function calls from your module, have you considered where and how those symbols will be resolved? On AIX, some up-front planning will avoid much pain. For your dynamically loaded module, ensure that any out-of-module references will be satisfied by modules that are already part of the running process. Those required definitions could be part of the main application, or be provided by other modules that were dynamically loaded first. In any event, don't get the "cart before the horse" ensure that definitions are available before any references are made.

Deciding whether to use RTL

If you're porting an application to AIX, or starting a new development effort, you may be considering using runtime linking. For applications originally architected on other systems, you may find that they are so complex that you must use RTL on AIX. But if you're unsure, here are some points to consider:

1. Is there a tendency, in the code, to create more than one definition of a given symbol? Does the build process (on the other platform) depend upon command line ordering to select the required version of a symbol? If so, then RTL can assist you in getting the right answer on AIX, and avoid the need to rearchitect.

2. Is the application governed by runtime determination of function? I.e. is the system so flexible that function is determined and loaded at runtime, and additional function uses whatever is available based upon a dynamic configuration? (Think scripting that loads various components, or implementations of interfaces, and accomplishes work using the available features.) A highly dynamic system may best be implemented using RTL.

3. Is the application user-extensible? This might be an indicator of the need for RTL.

4. Is the code for the application well organized and separate? Does each subsystem or component provide a well-defined set of inputs and outputs? If so, you may find that the traditional AIX facilities are a good

match and that RTL can be avoided.

As you plan your port to AIX these questions illustrate the types of issues you'll want to consider during your initial phase.

Section 4. slibclean and file permissions

If you have read my tutorial on using the kernel debugger to investigate module dependencies, you have a good understanding of the AIX shared library regions. In both the 32-bit and 64-bit address spaces, shared modules are considered a global resource, and the shared library regions are commonly accessed by every process. AIX also adheres to the idea that once a module is loaded, it stays loaded, for use by any/everyone henceforth. Just because your program stops using a module doesn't mean that it is automatically removed from this region. Clearly, though, there are times when modules are loaded "publicly", or in the global shared library region, and you don't necessarily want to keep them there. AIX comes with a tool for cleaning up: slibclean. The slibclean command will do the following:

1. Examine the 32-bit shared library region and unload modules that are no longer in use by any 32-bit process.
2. Examine the 64-bit shared library region and unload modules that are no longer in use by any 64-bit process (64-bit systems only).
3. Traverse the kernel loadlist and remove any unused kernel namespaces.

The downside is that slibclean, by default, can only be run by the superuser. If you keep in mind that shared library regions are intended to be persistent, public resources, you can understand that it's not expedient to run the slibclean command often. Technically, cleaning up the shared libraries requires a lock on this global resource. This exclusive operation prevents modules from being loaded while slibclean runs, which means that exec and load operations can be delayed. This may not be an issue for your specific environment, but understanding the behavior can help you avoid potential problems.

You may have already jumped ahead to the next question: what about managing shared modules during development? There are two points to keep in mind:

1. The public loading of modules can be controlled by file permissions. If your intent is to build a module, test and debug, modify code and rebuild and retest, then it is suggested that you set your umask to automatically remove read-other permission on your files. If a module's disk file does not have read-other permission, it is not a candidate for public loading. If your files meet this condition, they will be privately loaded (read my kernel

debugger tutorials for more information on where modules get loaded in the address space). Modules that are privately loaded remain in use while the process is running; when the process ends, the modules are unloaded. This behavior is well-matched to the edit-build-debug cycle of application development. Once the module is "ready for prime time" enable read-other permission, and put the file in an accessible directory, to allow public consumption of the module.

2. If it is necessary to allow normal users the ability to clean up the shared library segments, you can always turn on the set-UID bit of the slibclean command. You'll also need to allow your users the ability to read the file. As superuser:

```
$ cd /usr/sbin
$ chmod go+rx slibclean
$ chmod u+s slibclean
$ ls -l slibclean
-r-sr-xr-x   1 root  system  1860 Mar 17 17:13 slibclean
```

Section 5. Wrap-up

Summary

Congratulations on completing the second tutorial in this series. Your increased understanding will serve you in developing applications that exploit runtime linking, and in building modules that behave as expected in both traditional and RTL-enabled AIX applications. You also understand how best to utilize the slibclean command, as well as its behavior and impact upon the system.

In AIX Linking 103 I'll continue to detail linker options, plus discuss some new commands introduced in AIX 5.2.

Resources

Learn

- Information on the LIBPATH environment variable and some useful linker options can be found in the Linking 101 tutorial at http://www.ibm.com/developerworks/eserver/tutorials/aix_link.html.
- Information on the AIX 32-bit and 64-bit address spaces, plus behavior of the AIX loader, can be found in the tutorial "Using the AIX kernel debugger to investigate module dependencies".
- For more comprehensive coverage, read AIX Linking and Loading Mechanisms.
- The AIX5L technical manuals fully discuss the ld command, dlopen() and load() system calls, and the XCOFF file format.
- AIX and UNIX: Want more? The developerWorks AIX and UNIX zone hosts hundreds of informative articles and introductory, intermediate, and advanced tutorials.
- Stay current with developerWorks technical events and webcasts.
- Want more? The developerWorks IBM Systems zone hosts hundreds of informative articles and introductory, intermediate, and advanced tutorials on the eServer brand.

Get products and technologies

- Build your next development project with IBM trial software, available for download directly from developerWorks.

Discuss

- Participate in developerWorks blogs and get involved in the developerWorks community.

About the author

Gary Hook
Gary R. Hook is a senior technical consultant in the Solutions Development Group at IBM, providing application development, porting, and technical assistance to independent software vendors. Mr. Hook's professional experience focuses on Unix-based application development. Upon joining IBM in 1990, he worked with the AIX Technical Support center in Southlake, Texas, providing consulting and technical support services to customers, with an emphasis upon AIX application architecture. Now residing in Austin, Mr. Hook was a member of the AIX Kernel Development team from 1995 through 2000, specializing in the AIX linker, loader, and general application development tools.