AIX linking 101

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Tutorial roadmap

Tutorial roadmap

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 to make effective use of the `LIBPATH` environment variable
- How `LIBPATH` interacts with the linker
- How to direct the linker when creating loader information in a module
- How the kernel finds dependent modules
- How to effectively use certain linker command line options

What is this tutorial about?

This tutorial is for developers who write or port applications to AIX and who want a quick introduction to the most commonly used features related to the linker and loader. The first of a series of practical tips and techniques, this tutorial focuses primarily on the library search path. Watch for more tutorials on a number of linker options.

Tools needed

The features discussed in this tutorial were initially made available in AIX 4.2. Later levels of the operating system provide behavior compatibility, including 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. Introduction

Introduction

I've been working with AIX for twelve years, five of them in AIX kernel development. My primary interest has been the AIX linker and loader and application development. I've spent a great deal of my time helping others learn how to use the tools, answering questions, solving problems, and generally trying to transfer skills to those who need them.

Looking back, I can count a lot of common tips and techniques I've explained -- and others have explained -- in repetition to large numbers of AIX programmers, but these things have never before been written down in plain, simple statements. So that's what we're doing in this tutorial -- teaching tips and techniques common to many frequently encountered situations facing AIX programmers. Use it as a cheat sheet.

You might be an application developer new to AIX, or you might be quite experienced with AIX. Take this tutorial before you read the technical manuals, or take it as a summary after you review the manuals. Or use it as a companion to other, more exhaustive papers (such as *AIX Linking and Loading Mechanisms*). This tutorial doesn't replace or conflict with those more extensive sources; instead it coalesces, summarizes, and puts into context some of the relevant technical points about the linker.

Warning: This tutorial is full of opinions. They're opinions based upon my experience in supporting customers. They're my opinions about code development and architecture and about the problems most frequently encountered by both new and experienced AIX application developers. I would be most grateful if any errors or inconsistencies were brought to my attention.

Section 3. Library search path

Where is the library search path found?

One of the most common concerns is the use of library search path information. The AIX environment variable `LIBPATH` correlates to the `LD_LIBRARY_PATH` variable on many other systems (for example, Solaris and Linux), and to `SHLIB_PATH` on HPUX. The `LIBPATH` variable also affects the link-editor in the same way as on other systems. If `LIBPATH` is set in the environment, the value is read and respected by the ld command. If a link operation occurs, either directly or driven by one of the compilers (`xlc, xlC, xlf`, etc) then the value of `LIBPATH` is used to search for dependent libraries (the `-L` option) **and** the contents of `LIBPATH` are stored in the resulting module. In this case, the contents of `LIBPATH` prepend the default library search path, which is `/usr/lib:/lib`. This might not be the behavior you intended, so once the module is built, you should view the library search path contained in the module by using the dump `-Hv` command. For example, consider this loader header

```
                    ***Loader Section***
                       Loader Header Information
VERSION#         #SYMtableENT     #RELOCent        LENidSTR
0x00000001       0x00000103       0x000002a9       0x000000ac

#IMPfilID        OFFidSTR         LENstrTBL        OFFstrTBL
0x00000005       0x00003854       0x00000d7b       0x00003900


                          ***Import File Strings***
INDEX  PATH                              BASE                MEMBER
0      .:../../lib/AIX:/usr/lib/threads:/usr/lib:/lib
1                                        libpthreads.a       shr.o
2                                        libc.a              shr.o
3      /opt/lotus/notes/latest/ibmpow libnotes_r.a
4      /opt/lotus/notes/latest/ibmpow libvsapi.o
```

Figure 1: An example of dump `-Hv` output

Look at entry `0` of the Import File Strings section. This information will be used by the loader to find dependents, in addition to the contents of `LIBPATH` at runtime and the search path in the executable (read on for more detail). This particular header, built on an AIX 4.1 system, also illustrates some of the problems that are addressed by the features added to the linker in AIX 4.2. More on this topic later.

To further complicate the situation, `LIBPATH` is ignored at link time if the command line contains any use of the `-L` option. In this case, the paths specified by one or more `-L` options are concatenated in the order they appear on the command line, and then prepended to the default library search path. The resulting composite path specification is stored in the loader section of the constructed module.

`LIBPATH` at runtime (exec)

Regardless how a module was constructed, LIBPATH can be used at runtime to direct the loader (a component of the kernel) to search in specific directories for dependent modules. The contents of the environment variable are prepended to the library search path found in the executable; this composite is then used at exec time to find dependents of the main program.

But additional rules apply. It's worthwhile to refer to the AIX 5.1 documentation, where the rules for finding modules at *dynamic load time* are listed in the manual page for the load() function. But exec time is not quite the same, so let's start with the rules for finding modules when a program is exec'ed. The possible components of the search path are:

1. The value of LIBPATH, if it is set in the environment.
2. The library search path contained in the main executable.
3. The library search path of the module being loaded.
4. The library search path contained in the loader section of the module whose immediate dependents are being loaded. Note that this per-module information changes when searching for each module's immediate dependents.

These elements are used, in order, to construct the library search path used when hunting for dependents.

LIBPATH at runtime (dynamic load)

When dynamically loading modules, the rules are:

1. If the L_LIBPATH_EXEC flag is set, the library search path used at exec-time.
2. The value of the LibraryPath parameter if it is non-null. Note that a null string is a valid search path which refers to the current working directory. If the LibraryPath parameter is NULL, the value of the LIBPATH environment variable is used instead.
3. The library search path contained in the loader section of the module being loaded (the ModuleName parameter).
4. The library search path contained in the loader section of the module whose immediate dependents are being loaded. Note that this per-module information changes when searching for each module's immediate dependents.

(You can find this information in the AIX documentation at

http://publibn.boulder.ibm.com/doc_link/en_US/a_doc_lib/libs/basetrf1/load.htm#HDRSDM12345
) To find the specifically named module, components 1 and 2 are used; all four components are used, in order, to find dependents of the named module.

LIBPATH summary

You can see that LIBPATH (with the exception of a programmatic specification in the load() function) takes precedence over information that is part of a module.

Finally, note that when a program requires the gaining of privilege, the LIBPATH environment variable is cleared. For security reasons, a process designed to run with an alternate effective user or group ID should find dependent modules only from trusted locations that are embedded within the application. In most cases, for example, an application that runs with superuser authority will usually have its dependents located in the /usr/lib directory.

LIBPATH problems at runtime

One of the most common problems encountered by users is the inability to find a dependent module. The error message produced by the kernel looks something like (on a 4.3.3. system)

```
$ ./main
exec(): 0509-036 Cannot load program ./main because of the following errors:
        0509-022 Cannot load module ./foo.so.
        0509-150   Dependent module bar.so could not be loaded.
        0509-022 Cannot load module bar.so.
        0509-026 System error: A file or directory in the path name does not ex
```

By reading this message you can figure out that the system found the main program and the first module, but it was unable to locate bar.so. The action required by the user is to use dump -Hv to see what main contains for a library search path, echo $LIBPATH in the shell to see if/how it is set, and generally figure out where the bar.so module is supposed to be found. In many cases the name of the missing module provides some indication of its purpose and supposed location. Failing that, the user can always seek help from the system administrator.

On earlier levels of AIX, the message displayed when a direct dependent could not be found was the same as that shown when a *dependent of the dependent* couldn't be loaded. This led to the false conclusion that the immediately dependent module was the point of failure, when in fact it was often found that the actual problem was farther down the dependency tree. What should you do? You must (1) carefully read the error message to glean as much information as possible, and then (2) use dump -Hv, starting at the main program, and work your way down through the layers of modules to figure out where the actual failure is occurring. At some point way down there, you'll usually discover a missing module, or find that the required module is in a directory that is not part of the library search path.

The -blibpath option

When you're developing applications on AIX and deciding upon the architecture of the program files, a common situation is that the build environment for the application

doesn't necessarily reflect reality as it applies to the installation of that application on a production machine. Often, the manner in which shared libraries (modules) are constructed, how they reference one another, and how they are referenced by the applications themselves can be trivially disorganized, or it may be grossly complex. One issue is how a build environment can be used in a flexible manner during a link operation, yet that build environment structure is not evident in the end product. And this applies most often to the library search information that the linker stores in the constructed module.

Recall from above that, between the `LIBPATH` environment variable and the use of `-L` options, a number of paths may be concatenated and stored in the linker's output file. What is needed is a way to do whatever it takes to generate a proper ld command line, yet have the resulting module not contain any information that likely has no applicability to the production environment.

What to do? Use the `-blibpath` option. This useful option allows the specification of an exact library search path that should be stored in the output module. Regardless of the search paths used when link-editing, the final product will contain the string that follows the `-blibpath` option. For example, to force the default library search path into a module, a (portion of a) command line would look like this:

```
... -L/lib/foo -L/lib/bar -blibpath:/usr/lib:/lib ...
```

The resulting module would have a basic, default search path as entry 0 in the Import File Strings section of the loader header (dump `-Hv` <filename>).

Finally, examine the loader header shown in figure 1. You can see that the search path (in entry 0) contains a relative path. Clearly, this path was used when this program was linked; on an end user's machine, however, it is meaningless. Aside from creating extra work for the loader, which must try to access the specified directory when conducting its module search, the path serves no purpose. In this case, the `-blibpath` option would have been very convenient.

Absolute paths vs. relative paths

No doubt about it: relative paths are evil, at least in the search for dependent modules. The primary problem with relative paths is that their use depends upon a context (that is, the directory from which the reference is made). As most applications are intended to be run by a user in a completely arbitrary directory, any relative path is going to based from a location that was unforeseeable when that application was constructed.

To be fair, though, relative paths do come in handy in build environments. Even the environment in which AIX is built takes advantage of relative paths during link operations. But that doesn't make them practical at runtime. So you are strongly encouraged to take advantage of the `-blibpath` option to circumvent any unpleasant side effects that may result from using relative paths with `-L`.

At runtime, in `LIBPATH`, it's just inadvisable to use relative paths. I'm unaware of any situation where an absolute path could not be used just as effectively, and with less

confusion.

Section 4. More linker Options

The -bnoipath option

Another very useful option is -bnoipath. This option removes path information from dependent modules listed in the loader section of the output file. Why is this interesting? A build environment can take many forms. Often, the linker's -L option is used to list a search path; at other times, dependent modules are specified by providing a path to the file. This implies that archive files and/or loadable modules may be listed on the command line in the same fashion as object files. It's not wrong or right, just appropriate to a particular situation. So, where -L and -blibpath together can be used to find dependents for a link operation, so can naming the files explicitly and using -bnoipath to trim the path information.

Look again at figure 1. In this header there are two dependents that contain path information, entries 3 and 4. The problem with this construct is that at runtime those dependent files (libnotes_r.a and libvsapi.o) must be located *in precisely the directory specified in the loader header*. No use of LIBPATH will change that fact. If the files aren't found in /opt/lotus/notes/latest/ibmpow, this program won't run.

Using the same link command to build this program, but adding -bnoipath, would produce an Import File Strings section like that shown in figure 2.

```
                    ***Import File Strings***
INDEX  PATH                                BASE                MEMBER
0      .:../../lib/AIX:/usr/lib/threads:/usr/lib:/lib
1                                          libpthreads.a       shr.o
2                                          libc.a              shr.o
3                                          libnotes_r.a
4                                          libvsapi.o
```

Figure : The loader header after using -bnoipath

With this configuration, the four dependent modules could be located anywhere and LIBPATH would be utilized to find those dependents, resulting in much greater flexibility when installing and running this application.

The -T and -H options

The -T and -H options were necessary on AIX 3. No more. AIX 4 introduced a linker that could neatly pack the sections of the object file together, and generate loader information that allowed the loader to map portions of the file neatly into memory, all without requiring the sections to begin on certain boundaries.

Yes, the options have their place; for example, they are often required when constructing applications for embedded systems. But generally speaking, they're no longer needed.

The -e and -bnoentry options

On AIX, every module has an entry point, including shared (or "loadable") modules. The linker understands that the default entry point is "__start", and for executables, this entry point is defined in the crt0.o file that is included automatically by the compiler. Therefore, the linker's requirement for this symbol is satisfied without any special action on the part of the user.

When building a shared module, however, you have a choice. As a parallel to other systems, a module can be constructed without an "entry point;" the -bnoentry ld option would be used. The resulting module just presents a bunch of symbols (its API, or programming interface) representing the code and data within the module. In addition, on AIX the module can have an entry point, similar to an executable. This entry point can serve the same purpose as the one for a main program: if the module is dynamically loaded via the load() function, the return value from load() will be the entry point. If the entry point of the module is a function, then the return value can be cast to a function pointer, and the entry point called directly. Another example is using a data location as the entry point. If the parent program understands what data is contained within the structure referred to by the return value from load(), it can then cast the pointer and de-reference it to access the data in the loaded module.

Whether a module has an entry point has no bearing on any other use of that module. The module can still be used as a dependent in a link operation, or dynamically loaded via dlopen(), load(), and so forth. Just note that the entry point is only accessible programmatically by using load().

When building a shared, or loadable, module, and an entry point is not required, use the -bnoentry option. If you are building your modules by invoking the compiler (for example, cc) rather than ld, this will prevent the inclusion of the start-up code in the generated module. This will avoid duplicate definitions of global symbols in your module, and therefore avoid potential confusion between modules and the main application. More on this subject in a future tutorial.

Section 5. Summary & References

Summary & References

Summary

Thus ends the first in a planned series of tutorials that provide simple, practical suggestions on the use of the AIX linker and loader. Watch for the next tutorial, AIX linking 102, where I'll offer some practical suggestions on the most common issues surrounding runtime linking, slibclean, and whatever else space permits.

References

For more comprehensive coverage, see the article *AIX Linking and Loading Mechanisms*.

The *AIX5L technical manuals* fully discuss the ld command, *dlopen()* and *load()* system calls, and the XCOFF file format.

Section 6. About the author

About the author

Gary R. Hook is a senior technical consultant 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. You can contact him at *ghook@us.ibm.com*.

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.