Advanced Linking

Gary R. Hook
hook@vnet.ibm.com

AIX Systems Center
Dallas, Texas

Overview

- ☞ Review of Terminology
- ☞ Application Organization
- ☞ System Linker/Binder
- ☞ Kernel Loader
- ☞ Linker Command Line

Notes

This presentation is a discussion of some of the more esoteric aspects of the AIX Version 3 linker, or binder, and loader. By way of discussion, it is perhaps advantageous to approach the capabilities of the components through illustrations. These illustrations elaborate on the techniques used to solve various problems that have been brought to the attention of National Technical Support.

The first section of the presentation, once past the overview, is a brief discussion of application organization. This discussion will point out some potential problem areas that form the basis for more elaboration further on in the presentation. Following that is a brief rehash of binder terminology, then a series of "case studies". Each case study will explore a different problem that has been encountered.

After a series of case studies has been presented, some of the linker command line options will be explored.

For reference purposes, the command **ld** is the system linker. The linker processes command line arguments and hands instructions to the binder program **bind**, located in the **/usr/lib** directory. For the purposes of this presentation, the terms "linker" and "binder" are interchangeable.

Case Study Structure

☞ Problem statement

☞ Exploration of the nuances of that problem

☞ A description of how we might approach a solution

☞ Details of the solution

Notes

Our case studies will consist of:

✓ Problem statement

✓ Exploration of the nuances of that problem

✓ A description of how we might approach a solution

✓ Details of the solution

Each will explore the use of one or more of the binder/loader team to solve a particular problem.

Review of Terminology

☞ Reference

- ❑ The use of a symbol by some function or data structure

☞ Resolution

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

☞ Relocation

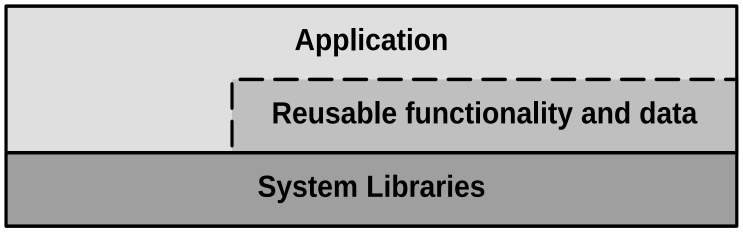
- ❑ The technique used to support the concept of address–independent programs

Notes

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 or dynamically loaded 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. This is accomplished by forcing all references to external functions to take place through the table of contents, or TOC. References to routines within the same object are handled via a function descriptor. It is through these function descriptors and the TOC that relocation takes place.

Application Organization



☞ Applications often contain code that can be classed as "reusable"

☞ Reusable portion is designed with an API

☞ The API defines the external access to the shared object

Notes

When designing an application to utilize shared objects, some thought must be given to the interface of the shared object. This is because the shared object is no different than an ordinary library with respect to programmatic usage, but does require additional care during program construction.

The reusable portion of any set of code falls into that category because there is a consistent programmatic interface to the code. This application programming interface, or API, often remains unchanged over the course of time, and thus can be used to describe the division between the main application and a shared object or objects.

Import and Export Lists

☞ The first line of an import list takes one of three forms:

- ❑ Anonymous import

  #!

- ❑ Specification of a shared object

  #!shr.o

- ❑ Explicit specification of a shared object

  #!/u/me/mylibdir/shr.o

☞ Export lists serve two purposes:

- ❑ Define the interface for accessing the shared object's functions and data
- ❑ Inform the binder which symbols to retain during the link phase

Notes

Import lists may take one of 3 forms: an anonymous import, a list that names an object to be loaded when found, and the full pathname of an object. In all three cases the file contains a list of symbols that are to be made publicly available to any referencing modules. The differences come from the use of each type of import list.

If an export list contains an object specification on the first line, that specification is ignored. This allows the same file to be used as both an export and import list. This feature will be explored in our first example.

One additional note: the first line of an import list begins with the characters **#!**(pound sign followed by an exclamation mark); occasionally, a module will contain symbols that begin with the character **#** (pound sign). If this is the case, the pound sign must not be the first character on the line, or the binder will think that the remainder of the line specifies the name of an object module. To sidestep this restriction, provide a blank space before any symbol beginning with a pound sign.

Case Study #1: Accessing Data Structures in Shared Objects

☞ Data can be specified within a shared object much the same as functions

☞ Often occurs with FORTRAN modules and common blocks in shared objects

☞ Data symbol resolution takes place at link time

- ❑ Shared object and main both reference the same symbol
- ❑ The links take place at different times
- ❑ Each “sees” its own copy of those symbols

Notes

Both functions and data structures may reside within a shared object. Since a reference to these elements is simply a symbol, they may both be treated in an equivalent manner with respect to the API.

For example, common blocks in FORTRAN modules are data structures created by the compiler. These structures exist within the defining module; it becomes the job of the linker to coalesce these definitions into a single datum.

Data Structures

☞ Modules referencing structure **`data1`** are

- ❑ Compiled
- ❑ Bound together to create **shr1.o**

☞ Other modules referencing structure **`data1`** are

- ❑ Compiled
- ❑ Combined to form the main application

☞ When the main portion is built, binding errors will occur due to the unresolved reference to **`data1`**

☞ If **`data1`** is defined within the main application, then *two* copies of the data structure will exist

Notes

How do we get the main application to connect to data structures located within an shared object?

For example, a data structure is defined and used within modules that are combined to form shr1.o. The data structure thus lives within that shared object, and references to the data structure from within that object are properly resolved.

Other routines are then compiled and linked to form the main application. This main application also references data1, but does not *define* it. The intention is for the main routines to access the data structure created within shr1.o.

To complicate matters, if the main portion attempts to explicitly define data1, then two copies of the data structure will exist within the executing application. This is a result of two properly compiled objects both completely and correctly defining their own copies of the same data structure.

One might jump to the conclusion that data of the same name would resolve to the same location in the data segment when all modules are loaded. This would not be the case due to the fact that all data from the main and shared portions will reside within the private data segment, but will occupy separate locations. There is a way around this behavior, which is discussed later in the presentation.

An Example in C

☞ Assume the following function

```
#include <stdio.h>

int     data1 = 5;

int     f1()
{
        printf( ”f1(): %d\n”, data1 );
}
```

☞ The symbols **`f1`** and **`data1`** are exported from the shared object

☞ The shared object is created with the command

```
ld -o shr.o f1.o -bE:shr.exp -bM:SRE \
   -lc -H512 -T512
```

Notes

Here we have **f1.c** containing the code for **`f1()`** and a definition of the data structure **`data1`**. An exports list is created which contains the symbols comprising the API.

The exports list for this object, in file **shr.exp**, will contain two symbols:

```
f1
data1
```

Note that we do not name an object on the first line of this file.

Finally, the object is created using a straightforward link command.

Please note that while this example is in the C programming language, this technique works with the FORTRAN language as well, and is especially applicable in the area of common blocks within shared objects. In fact, the technique came to light explicitly for providing the capability of accessing common blocks in FORTRAN shared objects.

An Example...

☞ Assume the following main routine

```
#include <stdio.h>

int    data1;

main()
{
       data1 = 3;
       printf( ”main(): %d\n”, data1 );

       data1 = 2;
       f1();
}
```

☞ The normal link command might be

```
ld –o a.out /lib/crt0.o main.o shr.o \
       –H512 –T512 –L. –lc
```

☞ Output from the application:

```
main(): 3
f1(): 5
```

Notes

Using common command line tactics, the main routine is compiled and linked to form the **a.out** executable. When run, however, the program produces erroneous output. We conclude that the two portions are not referring to the same copy of `data1`.

A Solution

☞ We need to tell the main application to use the data structures defined within the shared object

☞ The binder will utilize local definitions over import definitions

- ❑ This behavior is default using common command line semantics

☞ Import lists may also occupy a position on the command line

- ❑ Filename specifications
  - ➪ Object modules
  - ➪ Archives
  - ➪ Import Lists

Notes

We must somehow specify to the main portion of the application that the data structures of interest will be defined within a shared object. When that shared object is loaded (at program load time) then all references to the data structures will be properly resolved between the two portions of the application.

Common semantics of linker invocation are such that the pieces comprising the main application are listed on the command line, with shared objects and archive libraries listed after the regular object modules. In this case, the first *local* definition will override any imported definitions.

A Solution...

☞ An alternative exports file might be

```
#!shr.o
f1
data1
```

❑ Can be used as an import list

☞ The link command line

```
ld –o a.out /lib/crt0.o shr.exp main.o \
       shr.o –H512 –T512 –L. –lc
```

☞ Output from the application

```
main(): 3
f1(): 2
```

❑ Illustrates that both portions of the application are referencing the same data structures

Notes

If, however, we change the exports list into a form suitable for use as an import, we get the results shown in the first bullet. Note that the shared object is specified without a full path; this is one alternative to the build process.

The command line now shows the import list being specified explicitly on the command line. *Note that it is before the main routine.* This is the only location that will force the linker to override the local definition. The use of the **–bI:shr.exp** form will not provide the same results due to the order in which files and command line options are processed by the binder.

When the modified command line (shown) is used, however, the linker will override local definitions with the import version. This fact will be reported by **ld** during the execution of this command. Keeping in mind that the rule is first–come, first–served, the import version from the shared object will be the data structure used by the code within the main application. The last bullet shows the output of the application after the modified link command is used.

An alternative form of the export/import list and command line is (assuming the work is being performed within the **/u/me/mydir** directory)

```
#!/u/me/mydir/shr.o
f1
data1

ld –o a.out /lib/crt0.o shr.exp main.o \
        shr.o –H512 –T512 –lc
```

Here we have eliminated the need for the **–L** option in the link command. The choice of which is left as a philosophical exercise for the reader.

Case Study #2: Relinking a Shared Object

There are two aspects to this problem:

☞ Rebuilding shared objects during program development

- ❑ Use the relinking capability to speed program construction

☞ Replacing functions within a shared object that is part of the system

- ❑ Utilize a set of the developer’s functions to replace those normally found in system libraries
- ❑ Often desirable to provide a debugging form of the memory allocation routines.

Notes

One difficulty with shared objects is the inability to replace, on the “fly”, a function residing within the shared object. This requirement appears in two situations: building and debugging a shared object during the program development cycle, and replacing a routine that currently exists within one of the system libraries.

The first situation can be described as follows: in order to ascertain the execution performance and correctness of routines within a shared object, an application is separated into pieces, each of which becomes an object, except for the main portion of the application. Testing and debugging then takes place on the objects. As bugs are discovered and corrected, the new routines must be linked into the object.

The second scenario is more intricate. As an example, let us assume that a developer wishes to utilize a version of malloc suitable for debugging. This alternative form of malloc can be used to monitor heap usage and produce stack traces when memory use errors occur.

Further Details on the Problems

☞ Scenario #1

- ❑ Object must be completely rebuilt each time a change to one routine is made
- ❑ Ignores the relinkability aspect of objects under AIX

☞ Scenario #2

- ❑ References within an object are tightly bound
- ❑ System library cannot be modified by an average user
- ❑ Linker already understands how to locate the shared objects in the system library

☞ Shared objects are automatically viewed as “shared” by the binder

- ❑ How to access the actual contents of the object?

Notes

With respect to the first situation, every time a change is made to a routine in a shared object the entire object must be built using traditional construction techniques. As the shared object grows in size and complexity, the overhead required to rebuild the object increases.

In the second situation, there is no way for a symbol reference to be able to refer to an internal version of a routine and still be able to access an external version of the routine. Recall the first case study: references within a shared object cannot be changed without reconstructing the shared object. If this is the case, the problem becomes trivial. We must find a solution that allows us to utilize the existing object.

Further complicating the issue is the fact that references to the system library’s shared objects refer to the object **shr.o** in the file **libc.a** located in the directory **/lib**. Finally, the linker makes assumptions about the location of objects on the system; the loader then utilizes this information.

The final piece to this puzzle is the shared object itself. Normally, when linking to a shared object, the binder understands that the object is intended to be shared. It therefore only references the information within, but does not use the actual contents of the object.

Relinking a Shared Object

☞ Assume that an export list already exists for the shared object

☞ Be aware of references to other shared objects

☞ Pull in all referenced modules, shared or not

```
ld -r -bnso shr.o -o tmp.o
```

❑ Do not specify any of the system libraries in this command

☞ Relink the temporary object with the new modules

```
ld -o shr.o new1.o new2.o ... tmp.o \
   -bE:shr.exp -bM:SRE -H512 -T512 \
   -lc ...
```

❑ New modules must be specified first on the command line

Notes

The first assumption we will make is that an exports list already exists for the object of interest. A simple technique for producing this list is to use the following command

```
/usr/ucb/nm shr.o | egrep ' [BAD] ' | cut -c12- | \
     sed -e 's/^#/ #/' | sort | uniq > shr.exp
```

Note that there are spaces around `[BAD]` and before the # in the substitution string of the **sed** command. This command will produce fairly accurate exports lists for existing shared objects.

You will also want to get a list of archives and objects referenced by the shared object of interest. This is done using the **dump** command.

```
dump -nv shr.o | more
```

and view the "`***Import File Strings***`" section.

The next step is to get an actual copy of all the code of interest. This can be accomplished using the **-bnso** and **-r** options. This command should only list those objects for which you require the actual contents. Other objects that will be used as-is should not be referenced at this point. Name the output object with some arbitrary temporary name.

When binding in the new object modules, the export list will be used as well as any libraries that were originally required by the shared object. Note that the new objects must come first on the command line. In this way, they will override the definitions within the object **tmp.o**, and any references to the routines being replaced will be adjusted to refer to the new versions.

Case Study #3: Renaming a System Routine

- ☞ Code exists which calls a routine from a system library
- ☞ The calls to this system routine must be intercepted
- ☞ The system routine should be accessed via another name
- ☞ Application operates as–is on other platforms
- ☞ Source code cannot be modified

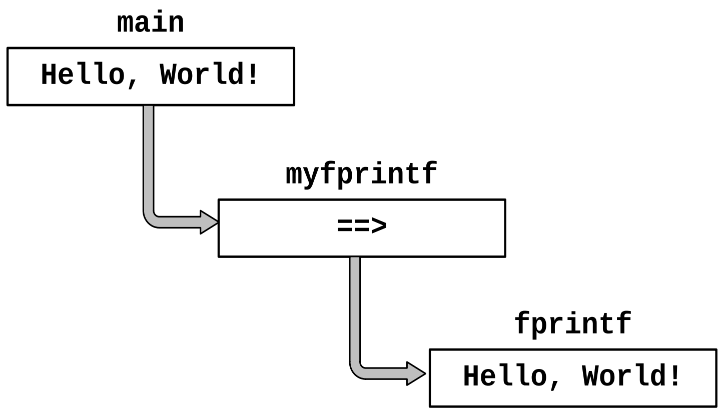
Notes

This problem relates to the issue of replacing any system function with a comparable one in an application. A “wedge” routine is supplied which is named the same, but performs some additional functionality before calling the real system call. One additional aspect of this problem is the probability of the code being portable between operating systems; the application should not have to be redesigned or source code modified for execution under AIX.

An Example of Replacing a System Routine...

- ☞ Calls to the system subroutine **`fprintf()`** should be intercepted

- ☞ A prefix is added to all text output through this function



- ☞ Output should look like

`==>Hello, World!`

Notes

As an example, suppose that all calls to the function `fprintf()` should be captured. The output is processed by placing a prefix before every group of text; the function arguments are then handed off to the real function.

An Example...

☞ The linker option **–brename:** is used

☞ True symbol names of functions are prefaced with a dot

☞ **main.c**

```
#include <stdio.h>

main( int argc, char *argv[] )
{
    fprintf( stdout,
       ”Arbitrary output from %s\n”,
       argv[0] );
}
```

☞ **fprintf.c**

```
#include <stdarg.h>

int fprintf( void *fp, char *fmt, ... )
{
    va_list arg;
    va_start( arg, fmt );
    sysfprintf( fp, ”==>” );
    vfprintf( fp, fmt, arg );
}
```

Notes

The rename option of the binder allows us to specify an oldname followed by a new name; the names are separated by a comma. Thus, the command line syntax looks something like

```
–brename:oldname,newname
```

True symbol names for functions in AIX objects are prefaced with a period. The name alone refers to the function descriptor, but we are interested in the actual function.

The sample code shows a call to the `fprintf()` function. Our version of the function outputs a simple prefix, then hands the function arguments to a function named `sysfprintf()`. During the link step, names will be changed to provide proper resolution.

An Example...

☞ Create an intermediate object

```
ld –r main.o fprintf.o –o tmp.o \
   –brename:.fprintf,.myfprintf \
   –brename:.sysfprintf,.fprintf \
   –H512 –T512
```

❑ Rename all references to **`fprintf()`**

➩ This includes our *definition* from **fprintf.c**

❑ Rename all references to **`sysfprintf()`**

☞ Build the application

```
cc –o main tmp.o
```

☞ Output of the application

```
==>Arbitrary output from main
```

Notes

The first step is to link all objects comprising the application. The system libraries are not included here as we will only manipulate references in our code. Since our application refers to functions in other libraries, we use the **–r** option.

All calls to the function `fprintf()` are transformed to an arbitrarily selected name `myfprintf()`. This renaming is also applied to the function from the file **fprintf.c**. After this step is completed, there are actually *no* references to the function `fprintf()`.

The second rename command takes the reference to the nonexistent function `sysfprintf()` and transforms it into a reference to `fprintf()`. This reference remains unresolved as this link phase completes.

The crucial point here is the order in which renaming is performed. We must have a point at which there are no references to the symbol being replaced. Once all existing references are changed, the “bogus” name for the function (`sysfprintf` in this example) is changed to the real name. Finally, this process must take place separate from any objects containing the actual desired functions (**libc.a** in this example).

The final link command takes the intermediate object and combines it with appropriate system libraries. The reference to `fprintf()` is resolved to the definition provided by **libc.a**

As can be seen, the output of the `fprintf` in `main()` is modified by our custom routine. While this example is fairly trivial, this approach works equally well with functions from any library or the kernel.

Case Study #4: Duplicate Symbol Names in C and FORTRAN

☞ Library routines are provided for two languages

☞ Under AIX, the true names of functions are uniform

- ❏ XLF does not add a trailing underscore to a symbol
- ❏ All FORTRAN symbols are mapped to lower case

☞ Uniform library routine names cause resolution conflicts

- ❏ The **`getenv`**/**`system`**/**`signal`** conflicts of last year are an example
- ❏ The IBM GL libraries are another source of resolution difficulties

Notes

Under AIX, interlanguage calling is simplified by the fact that symbol names between C, FORTRAN, and Pascal are uniform. Other systems historically have added an underscore to the end of FORTRAN symbols to distinguish them from C symbols. Our xlf compiler does not perform this function, by default; this simplifies housekeeping when creating multilanguage applications.

A difficulty arises when both C and FORTRAN (for example) access functions of the same name but residing within differing libraries or runtime environments. Since xlf does not mangle the symbol name, there is a conflict between the two libraries. Differing calling conventions between the two languages makes one dialect of the routine unusable by the other language without special coding considerations. One example is the conflict between the XLF runtime environment definition of the `getenv()`, `system()`, and `signal()` routines that occurred in the second and third quarter of 1991. This conflict appeared as programmers moved applications from other platforms to AIX. They would often make calls to `getenv()` or `system()` from both C and FORTRAN. Our library structure did not allow both language implementations to properly resolve to their respective versions due to the uniformity of the “true” symbols.

Another conflict arises in the area of GL programming. The GL libraries **libgl.a** and **libfgl.a** contain implementations of the GL graphics system with programming interfaces applicable to their respective languages. It is desirable to access GL routines from both C and FORTRAN within the same application, but symbol resolution during program linking produces the same difficulties as described above.

A Program Construction Strategy

☞ Shared objects allow us to implement symbol resolution at separate times

☞ Separate the C function from the FORTRAN

☞ Combine the C modules into a shared object linked to the appropriate libraries

- ❏ An exports list is required for the C modules

☞ Combine:

- ❏ the FORTRAN code
- ❏ FORTRAN libraries
- ❏ Shared object containing C routines

Notes

Using shared objects allows us to perform the correct symbol resolution at proper times. The first step is to separate the modules into groups based on the source language; our example will use C and FORTRAN.

Our main routine is written in FORTRAN; therefore, the C functions are selected to build a shared object. This conclusion is based on the fact that the native compiler corresponding to the language of the main function should be used to perform the final program construction. The C modules are combined into a shared object; an appropriate exports list will be required. This link step should provide proper resolution of all C–specific symbols during this step. To support the next phase of this process, the final location of this shared object is left as an exercise for the reader. We will assume it is in the working directory.

Finally, the remaining modules are combined with any FORTRAN–oriented libraries to produce the final application. It should not be necessary to add the C–oriented libraries at this point; all resolution to C routines should be accomplished through the shared. If, however, the FORTRAN code does directly access C library routines, the appropriate libraries may be added. The focus of this technique is to take the C *conflicts* and move them to a separate object.

An Example

☞ **main.f**

```
      program main
      call gef()
      call gec()
      end
```

☞ **gef.f**

```
      subroutine gef
      character*256   c
      call getenv( 'SHELL', c )
      write(5,10) c
10    format('SHELL=',A80)
      end
```

☞ **gec.c**

```
int gec()
{
    char *term = getenv( "TERM" );
    printf( "TERM=%s\n", term );
}
```

Notes

As an example, consider this source code. A call is made to the FORTRAN version of `getenv()` from the `gef()` subroutine; the C function `gec()` calls the C version of `getenv()`. Both modules are accessed from the main program.

This particular example is rather trivial. The current version of the XLF compiler understands certain functions that have caused problems in the past. The `getenv()`, `system()`, `abort()`, and `signal()` symbols, when accessed from FORTRAN, are modified into a format that avoids the symbol name conflict. This fix occurred at the 1.1.5 level of the compiler, but the principles involved in this solution are easily illustrated using this function.

An Example...

☞ Create the shared object

```
ld –o shr.o gec.o –bE:shr.exp –bM:SRE \
   –H512 –T512 –lc
```

☞ Create the main application

```
xlf –o main main.f gef.f shr.o –L$PWD
```

☞ Application output

```
SHELL=/bin/ksh
TERM=aixterm
```

Notes

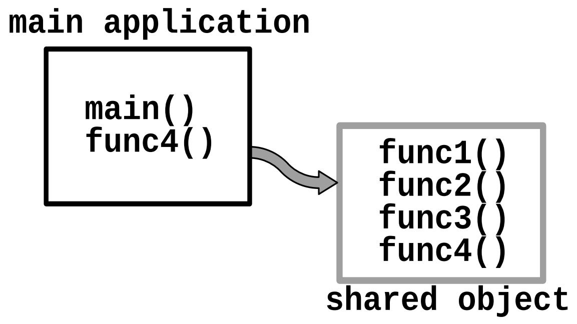
The shared object is built; don't forget to construct an exports list! The C library is added to this link; references from our C code are fully bound to the definitions within the C library.

The remaining portion of the application is built. Any FORTRAN references are resolved at this time; the resolution will be to the FORTRAN library. One can see that by separating the link steps, proper resolution can be enforced even though the libraries themselves might not be "cooperative".

Case Study #5: Module Load Order

☞ A shared object contains reusable functionality as well as replaceable routines

☞ The shared object should be designed such that **`func4()`** can be replaced by a function of the same name within the main portion



☞ Functions arbitrarily utilize replaceable functions

❑ **`func2()`** calls **`func4()`**

Notes

For this example, let's assume that we have designed a shared object to contain a certain number of reusable functions. In addition, this shared object contains functions that may be replaced by functions of the same name in the main application.

As an example, suppose we have created a shared object containing four functions. Given the behavior of the binder, all references to `func4()` within the shared object will be tightly bound to the existing definition. At runtime, the function `func4()` in the main application is the function that should be called by any routines within the shared object. An additional constraint is that the main application *is not required* to supply a version of `func4()` to the object. Therefore, a default copy still has to exist and be accessible, as the function will be used during execution. These two constraints are seemingly incompatible given the structure of objects in AIX.

Module Load Order...

☞ The loader utilizes the following:

- ❑ Object filename
- ❑ Library search path

☞ An application consists of one or more objects

☞ Each object has its own library search path

☞ This information is often uniform, since the default is **`/lib:/usr/lib:`**

☞ Search paths do not have to be identical across all objects that make up an application

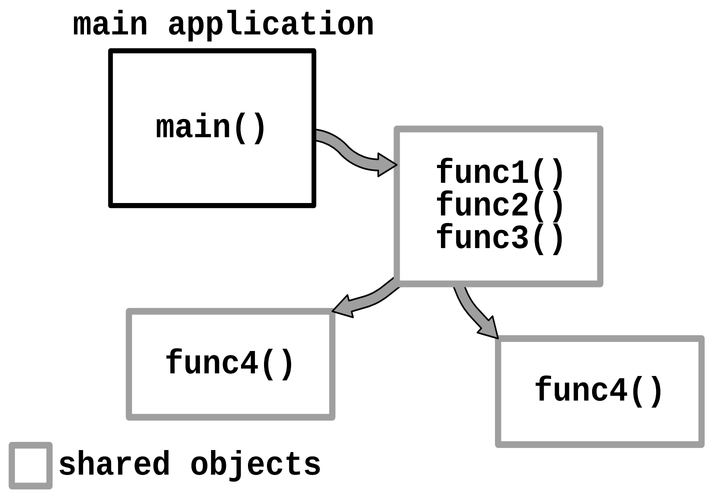
Notes

When we explore the functionality of the binder and loader, we discover that loader information is stored in every loadable object; i.e. every object that has been produced by **ld**. This information often is compatible across objects. For example, most objects contain information about the standard library search path **`/lib:/usr/lib:`** since they utilize the system library located in **`libc.a`**. But this information can change based on the object. Even modules comprising a single application can contain varying loader information. We can use this fact to reconstruct our shared object into a form more suitable to the situation.

Module Load Order...

☞ Split the shared object into two portions:

- ❏ The "reusable code" portion
- ❏ The replacable code portion



☞ Any replaceable function now resides within a separate shared object

Notes

Functions classed as "replaceable" in the shared object are extracted into a separate object. While this admitedly adds additional housekeeping to the application, it provides greater flexibility. The idea now is to construct the shared objects from the bottom up.

Functions such as `func4()` are grouped together to build an object. This object is built the same as any other shared object. For the purposes of this discussion we will call the object **repshr.o**. The remaining functions are then combined to construct the object **shr.o**.

Building the Objects...

☞ To build **repshr.o**:

❑ The exports file **repshr.exp** will be

```
func4
```

❑ The command to link the object is

```
ld -o repshr.o func4.o -bM:SRE \
   -bE:repshr.exp -H512 -T512 -lc
```

☞ To build **shr.o**:

❑ The contents of **shr.exp**

```
func1
func2
func3
func4
```

❑ The command to link the object is

```
ld -o shr.o func1.o func2.o func3.o \
   repshr.o -L/u/me/lib -bM:SRE \
   -bE:shr.exp -H512 -T512 -lc
```

Notes

Each object will have an exports list (no suprises here). The object **repshr.o** is built first; this makes the creation of the other object more straightforward.

Several points to note here include:

✓ The file **repshr.o** is located in a directory which is publicly available.

✓ The name **repshr.o** will be used by another object residing in a different location.

✓ **repshr.o** is built first to allow **shr.o** to be built in a manner which supports this scheme.

The other routines are combined to create **shr.o**. Note that we have two choices when linking:

✓ Specify **repshr.o** on the command line and use the **–L** option to provide the library search path modifier. Note that even though we might specify a path to the object module, only the basename of the file is retained.

✓ Use an import list that only names the object file **repshr.o** without specifying a full path to the object.

The point here is that, while we must name the shared object and provide an appropriate library search path, we must not irrevocably combine the two parameters. Thus, we name **repshr.o** as an object and add a search directory. The loader will, when attempting to load **shr.o**, search the specified directory for a file named **repshr.o** in order to resolve the reference to `func4()`. An examination of the loader section using the command

```
dump -nv shr.o
```

will show how this information is organized.

Building the Objects...

☞ The main application consists of

- ❏ The main code
- ❏ Functions designed to replace other (replaceable) functions
  - ➭ These functions are combined to create another object named **repshr.o**

☞ To build **repshr.o**:

- ❏ Again, **repshr.exp** will contain:

```
func4
```

- ❏ The command to link the object:

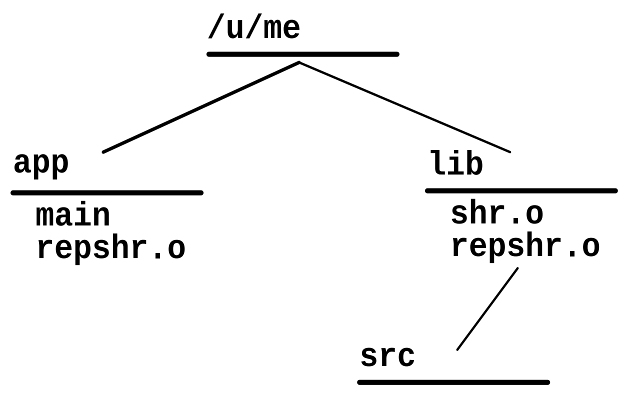
```
ld –o repshr.o func4.o –bM:SRE \
   –bE:repshr.exp –H512 –T512 –lc
```

Notes

The main application must also be split into two portions. Functions of the same category as `func4()` will occupy another object named **repshr.o**. This object will be built the same as the previous **repshr.o**. In fact, the same exports list may be used, as well as the same command line.

Building the Objects...

☞ The construction of the main application takes place in a different directory



☞ To build the main application, the command is

```
cc –o main main.o /u/me/lib/shr.o \
   –L$PWD –L/u/me/lib
```

Notes

The generic portion of the application resides in the **/u/me/lib** directory. The main portion is kept in **/u/me/app**. Thus, when the application is linked, we must specify a library search path to the application. The key point here, however, is that the library search path is different than the one specified for **shr.o**. Recall that for **shr.o** we listed the directory **/u/me/lib**; for the main application we will specify **/u/me/app** also, but *first* on the command line. This tells the loader to search for an object named **repshr.o** in the application directory first, then the **lib** directory, and finally in the standard locations **/lib** and **/usr/lib**.

Note, also, that this command line only references **shr.o**; it is not necessary to explicitly specify **repshr.o** at this point. The symbol table in **shr.o** will supply all the symbols to be referenced.

How does this differ from the file **shr.o**? When that object was constructed, the library path was listed as **/u/me/lib**. The result is two modules (**main** and **shr.o**) which contain differing search paths.

How does all this add up to a solution to the problem? It turns out that the loader, when bringing in an object, will use the library search paths *in the order in which they are found*. Thus, the loader loads **main**, which requires the object **shr.o**. The loader looks first in the paths specified by the object **main** while searching for an object named **repshr.o**. If one is found, it is loaded. In our case, this will be the file in the **/u/me/app** directory, the one containing the replacement functions. Thus, the original problem has been addressed.

What happens if an application is built which uses **/u/me/lib/shr.o** but does not supply any replacement functions? When **shr.o** is loaded, the loader will search for the file **repshr.o** in the paths specified in the main application; failing that, it will then look in the paths specified in **shr.o**. At this point it would discover an appropriate file in the **/u/me/lib** directory and load it. All symbols will be resolved and execution will begin.

Replacing Some of the Functions

☞ A function exists which is not replaced by the main application

- ❏ **`func5()`** is added to **/u/me/lib/repshr.o**

☞ Only one object with a given name can be loaded at a time

- ❏ The object **/u/me/app/repshr.o** does not contain a definition of **`func5()`**, but will be the first object found along the current search paths

☞ If the object **shr.o** cannot resolve the symbol **`func5`**, the load will fail

☞ An imports list is provided to **/u/me/app/repshr.o** during the link

```
#!/u/me/lib/repshr.o
func4
func5
```

- ❏ A corresponding exports list should be constructed which lists all of these symbols

Notes

That was a fairly trivial example. Let's complicate matters somewhat by adding another function `func5()` to **/u/me/lib/repshr.o**. This function should be used as–is by a certain application; it will be called from the main portion rather than the shared object **shr.o**.

The difficulty comes from the object load order: **/u/me/app/main** causes the object **/u/me/lib/shr.o** to be loaded. This object, in turn, desires to load an object named **repshr.o** from some arbitrary directory. If the object that is found (**/u/me/app/repshr.o** in the given scenario) does not contain the required symbol, the load will fail with an error message.

The solution is to get **/u/me/app/repshr.o** to understand the other symbol definitions. This is easily accomplished with an imports list that specifies the exact location of the alternative **repshr.o**. This list will provide alternative locations of symbol definitions; the same set of symbols is then exported from the object, but without naming the absolute path of the object.

Therefore, **repshr.exp** contains both

```
func4
func5
```

as the external interface to the module.

Replacing Some of the Functions

☞ The link command from within **/u/me/app** then becomes

```
ld –o repshr.o func4.o –bE:repshr.exp \
   –bI:/u/me/lib/repshr.imp –bM:SRE \
   –H512 –T512 –lc
```

❑ The import list is kept in the **lib** directory

☞ The object supplies local definitions for any defined functions

❑ The definition from **func4.o** is retained

☞ Pass–through references are supplied for any undefined functions

❑ The definition of **`func5()`** from **/u/me/lib/repshr.o** is passed through this object

☞ When loaded, **shr.o** references **repshr.o** which is found and loaded from **/u/me/app**

☞ References to **`func5()`** ultimately resolve to **/u/me/lib/repshr.o**, which is also loaded

Notes

When the object is linked together, the command line specifies an import list which will refer to every function coming from the generic module **/u/me/lib/repshr.o**. Any local definitions, such as `func4()`, will override the import definition due to the command line syntax being used (**–bI:** option instead of naming the import list specifically; refer to case #1).

At load time, then, **main** is loaded, causing **/u/me/lib/shr.o** to be loaded. This modules causes a search for **repshr.o** to take place; the object file is found in the **/u/me/app** directory. The pass–through reference also causes the object **/u/me/lib/repshr.o** to be loaded, but any references to `func4()` will bind to the first definition found, which is the one in **/u/me/app/repshr.o**.

The complete code for this example is provided in appendix A

An Idiosyncracy of the Loader

☞ Loading of dynamic objects occurs in one of:

- ❑ Segment 13, the shared text segment
- ❑ Segment 2, the process private data segment

☞ Destination is determined by file permissions

- ❑ Basic objects: the actual file permissions are used
- ❑ Archives: the permissions of the *archive* are used

☞ Holds true for both a shared object and the **`load()`** system call

Notes

When objects are loaded, under the current system, they will be placed into one of two places:

✓ The shared text segment (addresses beginning at 0xD0000000)

✓ The private data segment (adresses beginning at 0x20000000)

The only criterion used to determine the load location is the permissions of the file. A shared object (or dynamically loaded object) will only load into the shared text segment if the file permissions provide read access by "other". Note that execute access is not necessary.

The actual file permissions are the criterion. While this is a simple matter to determine for simple **.o** files, archive libraries become another matter. Under AIX Version 3, the permissions of the *archive* itself determine the load destination. This is in contrast to the permissions of the objects within the archive.

This behavior is consistent for both shared objects and dynamically loaded objects.

Programmatic Interface

☞ Dynamic loading and binding is provided by the following functions:

`load()`

- ➪ Loads an object, and returns a pointer to the entry point

`loadbind()`

- ➪ Causes explicit binding to take place between two objects

`loadquery()`

- ➪ Provides information about the list of objects that were loaded as part of the executing application

`unload()`

- ➪ Remove a dynamically loaded object from the process' memory

Notes

These four functions provide a high level interface to the kernel loader. The first function, `load()`, provides the capability of loading an additional module into an executing application. These object modules are referred to as dynamic object modules to differentiate them from shared object modules. The differences at this point in time are minimal, as they exhibit the same functionality. Also, the location at which they are loaded is dependant upon the file permissions: is read access provided to "other"? If global read access is provided, the object is loaded into the shared text segment; if not, it is loaded into the private data segment.

The `loadbind()` function provides access to the kernel symbol resolution service. The use of this function is explored in a following example.

When problems arise, the `loadquery()` function can be used to determine the source of the problem. It can also be used to access the list of modules that comprise the executing application. The AIX Calls and Subroutines Reference provides a description and example of the use of this routine; it is only mentioned here for completeness.

Finally, the unload() system call can be used to remove an object module from an executing application. If that module was loaded into the shared text segment, the system handles the reclamation of the used memory. If the object was loaded into the private data segment, that memory remains unusable except by another call to `load()`.

The `load()` Function

☞ Requirements for creating objects for dynamic loading:

- ❑ The name of the entry point, or
- ❑ An exports list

☞ The command line would look something like

```
ld -o dynobj.o f1.o f2.o ... -lc \
   -H512 -T512 -eEntryPoint
```

or

```
ld -o dynobj.o f1.o f2.o ... -lc \
   -H512 -T512 -bE:obj.exp
```

☞ The **`load()`** function allows the program to load a module

- ❑ And resolve all symbol references to the executing application
- ❑ Or wait until the object is explicitly resolved against another object

Notes

To create an object for dynamic loading, we use what is by now becoming a very familiar command line. The difference here is that the option **–bM:SRE** is unnecessary. We also have an alternative approach to specifying the external interface:

✓ An entry point can be named using the **–e** option. This dictates a single function which will be referenced by the loading program. This option is the same as providing an exports list containing a single symbol.

✓ An exports list can be named. With this option, various unrelated modules can be combined into a single object and loaded. Accessing these modules can be accomplished using a variety of techniques. More on this subject at the end of the presentation.

There would seem to be a conflict between these two approaches. How is the entry point determined when an exports list is used? In this case, the first data symbol in the list appears to be used as the entry point. While this is not documented, it does seem to be deterministic. It is advisable, however, to explicitly name the entry point. If there are no data symbols in the exports list, the entry point becomes undefined.

The `load()` function can be used to bring in, at runtime, various portions of an application as they are needed. Through the use of the L_NOAUTODEFER flag, objects can be loaded and explicitly resolved against other objects. This does not prohibit the main application from executing; the only potential pitfall is if the application attempts to access a symbol that has not been resolved.

One additional note: `load()` takes as an argument the name of the object file to be loaded. There is currently no way to specify a member of an archive. Thus, dynamic objects may not be combined into libraries.

The `loadbind()` Function

☞ The **`loadbind()`** function allows:

- ❑ Objects with unresolved references to be loaded and resolved against explicit objects
- ❑ Runtime selection of modules to load and reference
- ❑ Anonymous imports

☞ Various modules may be arbitrarily resolved against one another

Notes

It is possible to create objects that are not fully resolved. This is done through anonymous imports. When the object is created, not all functions or data symbols must be defined within the object; these definitions may be provided by the main application. This situation is similar to that found in our last example; the difference is that there is no default routine upon which to fall back.

This function, `loadbind()`, allows an application to resolve symbol references at runtime.

For example, an application can load two modules: `dyn1.o` and `dyn2.o`. The `loadbind()` function allows `dyn1.o` to be explicitly resolved against `dyn2.o`. The programmer can expand this technique to be as intricate as required; there is no practical limitation to the way symbol resolution can be accomplished.

Runtime Resolution

☞ An example of **main()**:

```
#include <stdio.h>
int     i1 = 1;
extern  int     f1();
int     main()
{
        loadbind( 0, main, f1 );
        i1 = 5;
        f1();
        printf( ”in main(): ” );
        printf( ”value of i1=%d\n”,i1 );
}
```

☞ Function **f1()** in the shared object:

```
#include <stdio.h>
int     f1()
{
        extern  int     i1;
        printf(”in shr/f1(): ” );
        printf(”value of i1=%d\n”,i1 );
        i1 = –3;
}
```

Notes

For example, suppose functions in a shared or dynamic object must access data structures that are defined and filled by the main portion of our program. Here we have a main program which defines data structure `i1`. The function `f1()` uses `i1` and resides in a shared object.

Runtime Resolution...

☞ An exports list **shr.exp** is constructed:

```
f1
```

☞ An import list **shr.imp** is provided:

```
#!
i1
```

☞ The shared object is constructed:

```
ld –o shr.o f1.o –bE:shr.exp \
   –bI:shr.imp –bM:SRE –H512 –T512 –lc
```

☞ Build the main application:

```
cc –o main main.c shr.o –L$PWD \
   –bE:main.exp
```

☞ The application output is:

```
in shr/f1(): value of i1=5
in main(): value of i1=–3
```

Notes

The exports list provides access to our function. An import list, in the anonymous form, indicates that the symbol `i1` is provided from somewhere; we just don't know where at this point.

The shared object is assembled. Note the use of the **–bI** option.

The main application is constructed. Here one detail must be attended to: the symbol `i1` must be exported. The file **main.exp** contains this symbol. Also, for the sake of simplicity, we assume that all the code for this example exists in the same directory. In more elaborate circumstances, attention would be given to the location of the shared object(s) and source files.

Finally, the application is executed. The output shows that the main routine can modify `i1`; the correct value is printed by the function `f1()`, which then makes its own modification. The final value is the printed by main.

Filenames

☞ Filenames on the compiler and linker command lines

- ❏ Object modules
- ❏ Archives
- ❏ Import lists

☞ Linker examines the contents of the file, not just the filename

☞ Shared objects could be named as an archive

- ❏ **shr.o** becomes **libmyshr.a**
- ❏ Allows the binder to avoid the overhead of accessing an archive

☞ Import lists can be named as '**.o**' files

- ❏ **shr.imp** becomes **shr_imp.o**

Notes

The binder is intelligent enough to recognize the type of file being referred to in the process of building an application. For example, one might assume that when I tell ld the name of an object file, "file1.o", it will trust me enough to believe that the file is indeed an object module. The same assumption would hold true for an archive file. It turns out that this is not the case. The binder determines the appropriate file type based upon the contents. Therefore, it is possible to change the name of a file for the benefit of the command line and the convenience of the build process. Let's explore some examples.

The first is one of creating and using shared objects. Normally, when a shared object is designed, the API is constructed, the code gathered together and linked to create an object. This object might then be placed within an archive, and the archive moved to a system–wide location such as **/usr/local/lib**. Now while there is nothing "wrong" with this approach, if the library in question only contains the single object, there is unnecessary overhead in accessing that object when linking. The loader will have the same overhead if the object is repeatedly required by executing applications and then released. The overhead is a result of opening the archive and locating the object within; both the binder and loader prefer to access the object without the intervening layer of the archive. To this end, then, an alternative might be take the shared object as before, but instead, change the name to that of an archive file. The linking semantics become simpler, and the actual creation and management of an archive to house the shared object becomes unnecessary.

A second example can be constructed from the case study #1 regarding the accessing of data structures within shared objects. In the example, access to a shared object's data structures was provided by preceding the objects on the command line with an import list specifying the true location of the data of interest. Note that this import list must fall first on the command line, but due to its name, requires direct use of the ld command. This implies much housekeeping and knowledge on the part of the average user which can be cumbersome. By renaming the import list to resemble an object module, the **cc** or **xlc** commands can be used to perform the link. Thus, this becomes a convenience issue.

Binder Command Line Options

☞ **–berok**

☞ Sample **main.c**

```
#include <stdio.h>
#include <sys/ldr.h>

main( int argc, char *argv[] )
{
    int (*FuncPtr)();
    extern int sub5();

    FuncPtr = load( ”sub.o”, \
      L_NOAUTODEFER, \
      ”/usr/lib:/lib:” );
    if ( FuncPtr == NULL )
    {
        perror( ”load failure” );
        exit( –1 );
    }
    sub5();
}
```

Notes

Several binder command line options warrant mentioning.

The **–berok** option can be used to force the production of an executable object, even though there may be unresolved symbol references. As long as the unresolved symbols remain unreferenced, the application will execute without mishap. In addition to this, this option can result in the following behavior:

When the application loads the object, any unresolved symbols in the main portion are resolved against the loaded object. This is default in the system. Any symbols that match those available in the new object will then refer to the definition in the object. This is how the symbol `sub5` is connected to the function `sub5()` within the object **sub.o**.

Note the option to the `load()` function: **L_NOAUTODEFER**. This option only applies to symbol resolution within the object being loaded. Resolution of symbols within **main** are still handled automatically. Therefore, the use of **L_NOAUTODEFER** or the standard flag **1** is irrelevant with respect to the main application. This flag only applies to resolution of symbols within the loaded object.

–berok Continued...

☞ Sample subroutine

```
#include <stdio.h>
int sub5()
{
    printf( ”inside sub5\n” );
}
```

☞ Build the dynamic object

```
ld –o sub.o sub5.o –bE:sub.exp \
   –T512 –H512 –lc
```

☞ Build the main application

```
cc –o main main.o –berok
```

☞ The output is

```
inside sub5
```

Notes

In this example, the `sub5()` function is very generic in nature; no surprises here. There is no requirement to do anything “magical” within the function.

The dynamic object is built using the techniques shown previously. The main application is then linked. Note that we do not specify **sub.o** on the command line; the main application handles accessing the object.

The important thing to note is the direct call to `sub5()`. When the main application is built, the symbol cannot be resolved; thus the need for the **–berok** option. At runtime, the object is loaded, and the symbol reference in `main()` is bound to the definition in the loaded object. This is all handled automatically by the kernel, and provides an alternative technique for access multiple functions within a dynamic object.

Command Line Options...

- ☞ **–bloadmap:<*filename*>**
- ☞ **–bnodelcsect**
- ☞ **–bnoobjreorder**
- ☞ **–bmaxdata:<*number*>**
- ☞ **–bmap:<*filename*>**

Notes

The **–bloadmap:** option can be used to produce a listing of binder activity that occurs during a link. This file will list binder commands and phases. Most importantly, symbol overrides are reported here. It would be very informative to utilize this option when working through the examples in this presentation.

Under AIX Version 3.1.7 or later, the default mode of the binder is **–bnodelcsect**. This is in contrast to earlier versions of the system. This option allows the binder to delete a symbol from an object module during processing. The previous default behavior was to remove an entire csect when a duplicate symbol was discovered. This occasionally produced an error message about missing symbols which were known to reside within the same module as the duplicate. Now, only the offending symbol is deleted; the csect, along with any other symbols it contains, is retained. This behavior is most often necessary when multiple definitions and multiple symbols occur within a module. If you are using an earlier version of the operating system, this option must be added to the command line.

Many programmers have requested an option which allows the user to bypass the object module reordering phase; **–bnoobjreorder** provides this functionality. This keeps the object code within an executable image in the same order as was specified on the link command line. This option can shorten the program development cycle, but the resultant application may experience performance degradation. This option also appeared at release level 3.1.7.

Another new feature of the binder under AIX Version 3.2 is the **–bmaxdata** option. This option allows the creation of a process private data segment greater than 256 MB. The actual method is to allocate up to 10 private shared memory segments to hold the data. This implies a maximum private data set of 2 GB, with an additional 256 MB for dynamic memory and the stack.

The **–bmap** option can be used to view the symbols, in address order, of an object. Since the binder will delete unused data structures, this option can be used to verify that a symbol is retained by an object.

Appendix A: Case Study #5 Sample Code

In the directory **/u/me/lib**:

f1.c
```
#include <stdio.h>

int     func1()
{
        printf( ″executing in shr/func1()...\n″ );
        func3();
}
```

f2.c
```
#include <stdio.h>

int     func2()
{
        printf( ″executing in shr/func2()...\n″ );
        func4();
}
```

f3.c
```
#include <stdio.h>

int     func3()
{
        printf( ″executing in shr/func3()...\n″ );
}
```

shr.exp
```
func1
func2
func3
func4
func5
```

f4.c
```
#include <stdio.h>

int     func4()
{
        printf( ″executing in shr/func4()...\n″ );
}
```

f5.c
```
#include <stdio.h>

int     func5()
{
        printf( ″executing in shr/func5()...\n″ );
}
```

repshr.exp
```
func4
func5
```

repshr.imp
```
#!/u/me/lib/repshr.o
func4
func5
```

The command lines are:

```
cc –c *.c
ld –o repshr.o f4.o f5.o –bE:../lib/repshr.exp \
    –H512 –T512 –bM:SRE –lc
ld –o shr.o f1.o f2.o f3.o repshr.o –bE:shr.exp \
    –L$PWD –H512 –T512 –bM:SRE –lc
```

In the directory **/u/me/app**:

```
main.c
#include <stdio.h>

main()
{
        printf( ”executing in main()...\n” );
        func1();
        func2();
        func5();
}
```

```
f4.c
#include <stdio.h>

int     func4()
{
        printf( ”executing in main/func4()...\n” );
}
```

```
cc –c *.c

ld –o repshr.o f4.o –bE:../lib/repshr.exp \
    –bI:../lib/repshr.imp –H512 –T512 –bM:SRE –lc

cc –o main main.o –L$PWD –L/u/me/lib /u/me/lib/shr.o
```

Output of the compiled and linked example:

```
executing in main()...
executing in shr/func1()...
executing in shr/func3()...
executing in shr/func2()...
executing in main/func4()...
executing in shr/func5()...
```