551

Managing programs and libraries in AIX Version 3 for RISC System/6000 processors

by M. A. Auslander

This paper describes the program and program-library management facility that has been developed for the AIX* operating system, Version 3, as implemented for the IBM POWER (Performance Optimization With Enhanced RISC) architecture. It provides run-time loading of libraries, symbol resolution with type checking, and relocation. In addition, the use of the loader to add programs to an already running process or to the kernel is offered. The advantages of these functions and the techniques needed to provide a usable and efficient realization are described. Particular attention is given to the special problems posed by very large programs, and by very small programs which use services from very large libraries.

Introduction

Traditionally, an executable program under a UNIX†-based operating system is self-contained. It is executed by reading its parts into predetermined memory locations and running it. Executable programs are built by compiling one or more source programs and using the **ld** command to combine them with programs from libraries. Library programs are selected to resolve calls in the original programs or in other library programs to functions or routines external to the calling program. Thus, a call to the **printf** routine results in inclusion of the library's implementation of **printf**. Since executing programs must contain actual machine addresses of programs and data, these *address constants* must be adjusted once the actual execution locations are known. This relocation is done by the **ld** command, and the resulting executable function will have a conventional fixed execution location.

The only remaining dependency of the resulting program is related to kernel services, which are represented in the program as system-call instructions, with each kernel service assigned a fixed system-call number.

This approach offers the advantage of low overhead at execution time, since loading a program involves only copying it to its predetermined locations. It is often possible to reuse the instruction portion of such a program for several executions, further reducing the cost of each. However, there are a number of drawbacks to this approach, which have become more serious as UNIX-based operating systems and their applications have grown. These include

- *Size of executable program* Since a program includes copies of all the other services (programs) it uses, its total size reflects the size of those services. This effect limits the use of such services.
- *Maintenance* Including a private copy of each service in each using program implies that changes to a service

*AIX is a trademark of International Business Machines Corporation.
†UNIX is a registered trademark of AT&T.

can only be installed by relinking and then redistributing the service to all using applications. This is of course not practical, and leads to very long lifetimes for old versions of existing services.

- *Distribution* A software supplier must either supply executable programs which contain particular versions of library services, or expect users to build final executable versions for themselves.
- *Name scope* The **ld** command treats all symbols from the user's program and from any libraries uniformly. In effect, each service included in the program is itself rebuilt from its components during the link. All interfaces and data in the service are exposed, and all external symbols in all services used by a program must be disjoint. Additionally, providers of a service risk having users become dependent on internal details of the service that were never intended to be part of its maintained interface. Programmers familiar with the various uses of the stream I/O package in *libc.a* are well aware of this phenomenon.
- *Kernel flexibility* The kernel interface consists of a set of system calls, using a machine-specific trap mechanism. These trap instructions become part of each using program. It is difficult to extend the system-call repertoire of the kernel, since a specific trap value must be assigned. Relinking is necessary to move a service into or out of the kernel.
- *Application packaging* No matter how large an application program is, it must be packaged as a single executable module. If this becomes impractical, the only alternative is to construct a multiple-process implementation, which involves major changes to the application structure.

In our design, we have replaced the traditional approach with one which can defer symbol resolution to program load time, and we have developed mechanisms for doing so efficiently for both large and small programs. We have provided for loading additional programs into the address space of a running process, and we have added a mechanism that allows for verification of compatible data types during link- and load-time symbol resolution. In the remainder of this paper, we discuss the overall strategy and describe some detailed techniques that are needed to make it work well.

In approaching this problem, there are trade-offs among strict compatibility with the past, performance at load time, and the provision of new function. Earlier approaches have either compromised the ability to replace libraries and to share data as well as code [1, 2], or accepted the performance penalty of reproducing **ld** semantics at load time [3]. Our approach may be characterized as accepting a modified symbol-resolution semantic at load time because we believe it is right and can perform better, even at the cost of some incompatibility with the past.

Strategy

At link time, programs are not (necessarily) combined with library programs. Rather, these libraries are loaded separately at load time and combined with the using program. To do this, **ld** records the names of the needed libraries so that they can be fetched at load time. Since we must be able to modify libraries without relinking the executable programs that use them, no internal details of the library can be introduced into the using program before load time. This implies that symbol resolution and relocation for library references must be done at load time. (Symbol resolution is the process of finding the definition of a needed symbol in a library; relocation is the process of updating address values in the loaded program to reflect the actual locations of the referenced symbols.)

Since we actually relocate address constants at load time, all the mechanisms described work uniformly for code and data. In particular, a using program can *import* the name of an external data structure from a library and use it just as if the library were bound with the using program. Since libraries can refer to each other or, in theory, even to the user program, symbol resolution and relocation must be done for the libraries as well as the program. (The alternative of assigning fixed, well-known, disjoint locations to all sharable libraries is clearly impractical.)

When possible, libraries are shared among many processes. The loader in fact places a copy of the library program into a shared part of the process address space, and uses the same copy for all processes. Of course, each process needs its own copy of the (read/write) data portion of the library routine. Each time symbols are resolved, either at link or at load time, a representation of the symbol type is included in the comparison. It is an error for symbols with the same name and different types to be present.

Imports and exports

It would be possible for load-time resolution to follow the same rules as link-time resolution. All the symbols of the program and libraries would be considered again at load time, and libraries would be searched for unresolved symbols.

We have chosen a more constrained technique, in which the only symbols available for load-time resolution are those explicitly *exported* by the libraries, and in which a library's internal symbol resolutions are never recomputed at load time. This approach follows the more modern preference for clearly distinguishing interfaces and implementations; it reduces the amount of work

done at load time, making high-performance loading practical. Finally, as we show below, by making the libraries independent of their users, much of the work of loading a library can be cached, reducing the cost of loading programs that use large libraries.

Table of contents

The final goal of load-time symbol resolution is to modify the address constants in the loaded programs to be the actual locations of the addressed data. Once resolution has determined the values, relocation updates the program. We expect programs to be large and use paging techniques to defer the actual reading of the program as long as possible. If address constants were spread throughout the program, then the program would in fact be completely read as a side effect of relocation, eliminating the advantage of "page mapped" loading. To avoid this, we introduce the *table of contents* (TOC). The idea is to gather all the address constants of a module together. This must be done even though the module consists of many separately compiled programs. The solution is to have each separately compiled program use register and displacement instructions to fetch needed address constants. The register is (usually) the conventional TOC register, which contains the origin of the running module's TOC. (There is one TOC for each linked collection of programs, or module.)

The linker collects all TOC definitions and constructs a single TOC with one instance of each address constant. It then sets the *displacements* of instructions which refer to TOC data to the correct value. (Relocation directory entries in each compiled program identify the TOC references.) When one program calls another in the same module, they both need the same TOC register value, so a simple branch-and-link instruction suffices. The called program, if it has static data, fetches the address of its own static data from the TOC. When one program calls another in a different module, it must set the TOC register to the TOC address of the called module. On return, the caller must restore its own TOC register value.

Of course, at compile time the compiler cannot anticipate whether a called program will wind up in the same or another module; this is not known until link time. One valid strategy would be always to load a new TOC value, but we decided to optimize the intra-module call by always using a simple, program-counter-relative, branch-and-link instruction with no TOC reload. If the called program is not part of the caller's module, the linker determines this and provides surrogate code in the caller which performs the needed TOC manipulation and register-contents-relative branch.

It turns out that the TOC provides even more advantages than might be apparent. A major portion of the "data" in many programs consists in fact of address constants. In a large modular program, there may be many copies of the address of each symbol. These are replaced by a single copy in the TOC. For small routines, the elimination of address constants often eliminates static storage completely.

Position-independent code

It would be impossible to share an executable program among several processes if the act of relocation were to modify the text (i.e., instruction) portion of the program. In this section, we describe the programming conventions that forbid this, and the characteristics of the POWER architecture hardware that make position-independent code efficient.

Local branches are by program-counter-relative displacements. Address constants are loaded into registers for use, and there are enough registers that such constants, once loaded, can be kept in registers. In position-independent code, the program cannot contain the addresses of its data, since the data may be in a different place for each execution and will certainly not be available before load time. Thus, the caller of a program must effectively provide the addresses of its data as well as the instructions for branching to the code. The TOC linkage convention provides for this by requiring that each program get its own data addresses from its TOC.

Procedure descriptors

To represent a "pointer to procedure," some value adequate to call the procedure must be used. As we have seen, this must include both the target-code address and the TOC address. But a procedure pointer itself must fit into a pointer value, so we decided to use a three-value descriptor to represent each procedure. A procedure pointer is then the address of such a descriptor. This descriptor contains the code address, the TOC address, and provision for an environment address for languages that need one.

Since the C language allows the comparison of procedure descriptors for equality, it is important that all uses of a library procedure "see" the same descriptor. For this reason, the procedure descriptor is always materialized in the data area of the module that contains the procedure, where it is treated like any other external data. Programs in this or other modules fetch an address constant to get the address of the descriptor, which is thus always the same. Since procedure descriptors contain addresses which must be relocated, the linker groups them together and places them adjacent to the TOC to reduce paging during relocation.

File names

When a module imports an interface from another, the using module will contain the file name of the used

module. This name is used by the loader to load the used module whenever the using module is loaded. To provide flexibility in the placement of shared programs in the file system, we have added a library-lookup mechanism which is similar to the path-lookup mechanism the shell uses for the main program. The file names in the using program may be either path names or base names. Base names are found by searching the directories in the **LIBPATH**. Of course, path names are used directly. The default value of the path string is stored in the module being loaded and is normally the same path used by **ld** in its library search. For exec, this can be overridden by the value of the **LIBPATH** environment variable. The **load** command has an explicit parameter for this purpose. By overriding the normal **LIBPATH**, the programmer can, for example, test a new version of a system library before installing it in the normal place. (The **LIBPATH** environment variable is ignored on **suid/sgid** calls.)

Once a file is loaded, it is kept open while in use. Thus, subsequent path lookups which result in the same open file are known to be the same file. The consequence of renaming or overmounting a file in use is that subsequent uses of the same file name are in fact seen to be a different file.

Sharing

In describing the **ld** command, we have indicated that shared modules are not copied into the output module. At load time, these modules are conceptually added to the text and data of the module. Two optimizations are applied during this process. First, if the "shared" module's access permissions allow universal reading, the module is copied into the shared-library region. Subsequently, other requests for the module can be satisfied by that copy. As long as a module is available for sharing, the file it came from remains open and "text busy," so that it cannot be modified. However, it can be replaced. If the module cannot be shared, a private copy is read into the process-private area.

When a module is placed in the shared-library region, a second optimization, pre-relocation, is attempted. The goal is to resolve and relocate this module once and for all. For this to be possible, all symbols imported by the shared module must come from the kernel or from other pre-relocated shared modules. When pre-relocation is possible, a target data location in the process-private area is assigned, and a copy of the data of the shared module is relocated as if it were at that location. At each use of this pre-relocated module, this data is copied to its target location, and no resolution or relocation is needed. A recursive algorithm is in fact employed so that sharable modules that use each other can be pre-relocated as a set. This succeeds as long as no module in the set imports symbols from a module which cannot be pre-relocated.

This sharing strategy cannot be implemented by an unprivileged loader, since the shared library area and the pre-relocations are shared by processes in different protection domains. Since the loader needed to be privileged in any case, we chose to make it a kernel service to further improve its performance.

Symbol resolution

Even though the imports/exports model reduces the number of symbols that must be processed at load time, libraries can still export hundreds of symbols. Thus, a linear search for each imported symbol would be inappropriate. Rather, as each module is loaded, a hash table of its exported symbols is built; the size of the hash table is chosen to be greater than the number of exported symbols. Thus, the import-symbol search time is linear in the number of imported symbols. When a shared-library module is pre-relocated, its symbol hash table is built and then reused at each use of the library.

Each module contains a list of the file names of the libraries used by the module. Each symbol is labeled with the specific library from which it came, and only that library is searched to resolve the symbol at load time. This further reduces the cost of load-time resolution, and also makes it possible to verify that a pre-relocation can be used without re-resolving the symbols of the pre-relocated modules. (If symbols were resolved by searching a list of modules, adding a module to the list or adding an exported symbol to a module would potentially change all symbol resolutions.)

An encoding of symbol type is carried with each symbol; this is a hash of the language definition of the symbol into a fixed, ten-byte field. Because the field has a fixed length (and is short) and because only equality is checked, the cost of this enforcement is low. The probability that two different types will match by accident after hashing is smaller than the probability that the machine will miscompare two values, and can be ignored.

Kernel name space

Rather than using system call or trap instructions for kernel calls, kernel symbols are imported in the same way as other shared-library symbols. The file name **/unix** is associated by convention with symbols exported by the kernel. When the kernel is built, some of its symbols are indicated as exports, just as for any other shared module. At system initialization, an exports hash table is built for these kernel symbols. In addition, some of the symbols can be designated as **syscalls**. At initialization, each **syscall** symbol is given the value of an interface routine, which issues a system call instruction that leads to the actual kernel entry point. Thus, to the using program, system calls are indistinguishable from other callable routines.

Archives

Traditionally, UNIX-based libraries are archives of programs, each the result of a separate compilation. A shared module is in fact a prebound collection of such programs. Thus, such a module could serve as a library, and we have used the two terms interchangeably above. However, the linker and loader also support archives, some of whose members may be shared modules. Remember that a prebound shared module contributes only its exported symbols to the link; whenever such a symbol is imported, its file name is recorded. If it is an archive member, its member name is recorded as well.

As an example of a shared library, consider **libc.a**. This library contains several programs which cannot be shared; these include **crt0.o** and **longjmp.o**. Each of these will be an archive member. In the simplest shared realization, all other object programs are bound into a single shared module, which is placed in a single archive member. **libc.a** consists of the private objects and the shared module. At **ld** time, only the few private programs and the exports list of the shared module need to be processed, speeding the **ld** step. At run time, the shared member of **libc.a** finds its way into the shared-library region and is pre-relocated. Thus, when a program is loaded, the loader need only look up the imported symbols in the already prepared hash table for this member and copy the pre-relocated data of **libc.a** into the process-private area.

Packaging

Packaging is the partitioning of a program into a collection of libraries and modules. It might appear that our design, which includes library names within the using programs, would make it difficult to change certain packaging decisions once made. In order to reduce this problem, the linker and loader allow for the "import of exports." A module or library which exports a symbol may in fact "produce" the definition by importing it from some other module. This allows the implementation of an interface to be moved, leaving an indirection in its old location so that pre-existing using programs will continue to work.

This mechanism can be used to move the implementation of an interface from one member of an archive to another. In particular, it allows an archive member to be broken into several pieces if it grows too large over time.

Another use of this mechanism is found in our treatment of **syscalls**. Whether a particular kernel facility such as **open** is in fact implemented in **libc.a** or in the kernel is a packaging decision which we would like to defer. **syscalls** are represented as being imported from **/unix**, which the loader takes as the name for the kernel services. However, we do not require each using module to indicate that its kernel imports came from **/unix**, as this would preclude ever moving part of a **syscall** implementation into the library. Rather, a shared member of **libc.a** will import (from **/unix**) and export these symbols. When a program is linked using **libc.a**, it will import the **syscalls** from this member. The implementation of these services can thus be moved by changing **libc.a**, without rebinding the using programs. (Because a hash table is used, these extra symbols do not increase the cost of resolving **libc.a** imports.)

Similar packaging changes are available to other library providers and users.

New system services

In addition to supporting exec, the loader provides several new services. The most important of these are kernel loading and the **load** system call.

- *Kernel loading*

In traditional systems, new device drivers, **syscalls**, or kernel services can only be added by rebuilding the kernel and rebooting. The kernel loader, however, provides instead for dynamic changes to the kernel without rebooting. The loading of these programs is similar to the user-level load. A loaded program can name imports from other modules, which are implicitly loaded with it. Additionally, if a loaded program exports symbols, these are added to the kernel name space. If any of these symbols are marked **syscall**, they will also be installed as new system calls, available to subsequently loaded user programs.

The **sysconfig** system call uses the loader to implement dynamic installation and replacement of device drivers, physical or network file system implementations, and other kernel mechanisms.

- **load**

The **load** system call allows a running program to add other modules to the process. The rules for loading these are similar to the initial exec load. The newly loaded module can import symbols from modules that have already been loaded, or from other modules that will then automatically be loaded. In addition, any module may import *deferred* symbols. These are left unresolved when the using module is linked. IF BINDDEFERRED, the default, is specified for the using module, each of these symbols will be automatically resolved by the first subsequently loaded module which exports it. If NOBINDEFERRED is specified, the **loadbind** system call must be used to cause resolution of deferred symbols in the using module from a specified exporting module that has already been loaded. The exporting module may have been loaded before or after the using module.

The **unload** system call marks previously loaded modules for unloading. These modules are removed from

the process image only when all uses can also be removed.

Complications

As with most new facilities, some complications are inevitable. **malloc** and the use of **_end** cause trouble when a program is loaded in pieces. We preserve **_end** as the last address in the data of the execed program. However, subsequent loads do not change its value. Thus, new programs using the **load** system call should not also depend on **_end**. We have also moved **brk** and **sbrk** into the kernel. Programs which directly examine the break value locations maintained by these services will fail. Also, programs must use **sbrk** correctly. They must not assume that sequential **sbrk** calls will always allocate sequential memory, because the loader competes with the user program for user memory space.

Programs which replace a **libc.a** service like **malloc** or **printf** cannot expect their replacement to affect shared-library uses of the service. If that is the intention, the using program must be linked to a private copy of **libc.a**. The linker provides an option for including private copies of all libraries, and warnings if a user program redefines a shared-library symbol.

The fact that sharable modules are kept in the shared library between uses exacerbates the text-busy problem. In many UNIX-based implementations a text-busy file cannot even be removed, let alone written.

We do allow text-busy files to be removed. However, the programmer must be aware that once a sharable program has been used, he cannot just rewrite it with a new version, but must remove the old version first. (Of course, removing a file while it is still in use is implemented correctly in that all existing uses continue to use the old value.)

Copying

In a number of places, we have indicated that values are copied. File contents are copied into both the process-private area and the shared-library region. Shared-library data is copied and pre-relocated. Pre-relocated data is copied into the process-private area. In fact, all these "copies" are normally done using well-known virtual-memory management techniques to delay actual copies until needed.

In addition, the virtual-memory manager and file system are expected to cache file contents in memory, even when the file is closed and later reopened. It is a goal of this design that repetitive execution of a given program will occur with *no* disk accesses in steady state.

Performance

We expect the performance cost of the shared-library load to be a small part of the total **fork/exec** idiom. In the end, this cost should be dominated by the number of unique data copies implied by its semantics. In order to reach this goal, we have used several techniques. The import/export semantics minimize the number of symbols which must be processed for each load. The technique of pre-relocation allows us to perform the expensive process of relocating large shared libraries in advance, caching the results, so that the cost of each load depends on the size and complexity of the using program, not the services it uses. Preliminary measurements indicate that these goals can be met.

Conclusions

The AIX Version 3 linker and loader for the RISC System/6000 processor have succeeded in implementing a high-performance, mostly transparent shared-library system. Programs are converted to shared-library use by linking them against shared versions of traditional libraries. We have found that using the exports construct, rather than reproducing **ld** at load time, causes little if any trouble even with existing programs. New versions of libraries can be built and used without relinking using programs. Finally, the execution overhead of the mechanism is small and linear in the number of symbols needed by the using program.

Thus, the ability of the linker/loader to provide traditional **exec** services seems evident. The usefulness of the new loader services awaits the test of time.

Acknowledgments

The first version of this approach was designed and implemented by a number of people as part of the CPr operating system at the IBM Thomas J. Watson Research Center. The TOC binder, without which much of this would be impossible, was first developed by Gregory Chaitin. Cliff Hoagland developed the AIX version of the TOC binder, which became the AIX **ld** program. Many of the ideas for the CPr loader and much of its final implementation are due to Albert Chang. Important contributions to the specification of the AIX version were made by Jack O'Quin.

References

1. "shlib command," *AIX Operating System Commands Reference*, Vol. 2, Order No. SC23-2081-1, September 1988, p. 939; available through IBM branch offices.
2. J. Q. Arnold, "Shared Libraries on UNIX System V," *Proceedings of the Summer 1986 USENIX Conference*, Atlanta, GA, June 1986, USENIX Association, Berkeley, CA, pp. 395–404.
3. R. A. Gingell, M. Lee, X. T. Dang, and M. S. Weeks, "Shared Libraries in SunOS," *Proceedings of the Summer 1987 USENIX Conference*, Phoenix, AZ, June 1987, USENIX Association, Berkeley, CA, pp. 131–145.

Received July 31, 1989; accepted for publication February 1, 1990

Marc A. Auslander *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Mr. Auslander received the A.B. in mathematics from Princeton University in 1963. He joined IBM that year at the Boston Programming Center and in 1964 became a member of the staff of the IBM Cambridge Scientific Center, where he did some of the earliest IBM work in paging computer systems. In 1968 he transferred to the IBM Thomas J. Watson Research Center to continue this work. He supported both university and industrial research through the publication of numerous papers and participation on the program committees of several operating system conferences. This early work contributed significantly to IBM's entry into virtual systems, and in 1973 Mr. Auslander wrote the key paper describing the IBM OS/VS virtual-memory operating system following the announcement of that system (now called MVS). After subsequently serving on the technical planning staff of the Research Division, Mr. Auslander became an original member of the research group that designed and built the first IBM RISC machine (the experimental 801) and the first RISC optimizing compiler. He later served a term on the IBM Corporate Technical Committee, and since returning to the Research Division has made important contributions to AIX Version 3 and other operating systems, to the AMERICA architecture, and to continuing development of RISC compilers. Mr. Auslander has received three IBM Invention Achievement Awards, three Research Division Outstanding Contribution Awards, and an IBM Outstanding Contribution Award for RISC compilers.