5 Program Loading and Dynamic Linking

5.1 System's Loader

The language processor is responsible for creating executable files that can be loaded by the system's loader. The loader binds object files and libraries together. Code in common object files and libraries need not be copied into the final executable file. Instead, these common modules are left in the original object files, and the executable file is updated to include the names and locations of these files.

Object files that contain common code are called shared objects. At load time, the system's loader will add all the required shared objects to the executable file's process image, a dynamic load of shared objects. Shared objects can be put into archive files, commonly called shared libraries.

The system's loader must be able to load shared objects to complete a process image at load time. It is not necessary for a process to use shared objects. However, if an application requires execution-time loading of shared objects, the system's loader must be able to do so. It must also be able to add shared objects to an already running process image and unload those shared objects.

5.1.1 Valid Load Modules

The system's loader will only load valid XCOFF load modules. (For more information, see Chapter 4, Object File Format). When loading an object module, the loader will obtain and validate information from various parts of the object file. The following fields must exist and conform to specifications listed in Chapter 4, Object File Format:

- **<filehdr.h>** structure field **f_magic**.
- **<aouthdr.h>**file
- **s_size** field
- **l_nsyms** field
- **l_nreloc** field
- **l_nimpid** field
- **l_istlen** field
- **o_algndata** field
- **o_algntext** field

Most of the information used by the system's loader during a load operation will come from the loader section of the load module. Figure 5−1 describes that section. (Figure 5−1 is a duplicate of Figure 4−3, reproduced here for convenience.)

Loader Section Header	32 bytes
External Symbol 1	24 bytes
External Symbol 2	24 bytes
...	
External Symbol *m*	24 bytes
Relocation entry 1	12 bytes
Relocation entry 2	12 bytes
...	
Relocation entry *n*	12 bytes
Import file ID 1	variable length
Import file ID 2	variable length
...	
Import file ID *o*	variable length
String table entry 1	variable length
String table entry 2	variable length
...	
String table entry *p*	variable length

Figure 5–1. Loader Section Structure

5.1.2 Making a Module Addressable

When the loader loads a module, the data contained in the module must be made addressable to the process. The process of making a module addressable is implementation-dependent, but the data contained in the module must have the following characteristics:

- The data contained in the text section of a loaded module must be read-only.
- The data contained in the data section of a loaded module must be read/write.
- The data contained in the bss section of a loaded module must be zero-filled and read/write.

When shared objects are loaded, each process must have its own copy of the read/write data (data and bss sections). A single copy of these sections cannot be shared among multiple processes. A single copy of the read-only data (text section) can be shared among multiple processes.

The access mode (permissions) of the object file determines the extent to which the module can be shared among multiple processes. The data can only be shared among processes that can access the object file in the file system. If a process cannot access the object file in the file system, it must not be able to access the loaded version of the shared object. Statically-linked applications are not ABI-conformant because application binary portability cannot be guaranteed from release to release. (The lib.c kernel interface is not part of the ABI.)

5.1.3 Shared Read-Only Objects

A special type of load module is a shared read-only object. A module is shared read-only if the **o_modtype** field of the **<aouthdr.h>** header contains the value RO, and the module does not import symbols from other modules. A shared read-only data module is one in which the data section for the module only contains read-only data. A single copy of this module's data must be shared among multiple processes. When shared read-only modules are loaded, the data contained in their data section must be made read-only.

5.2 Module Loading Overview

There are two ways to load a module for a process. One is the result of an **exec** system call in which the current process image is replaced with the new process image. The module loaded is specified in the **exec** system call. The other is as a result of the **load** system call in which the load module is added to the current process image rather than replacing it. In both cases there are basic operations involved in loading an object module, as are described in the following sections.

5.2.1 Loading Dependency Modules

The module specified in the **exec** or **load** system call is referred to as the *base module* of the load. Parameters to the **exec** and **load** system calls provide enough information to locate the object file for this base module in a file system. First, the loader makes the base module addressable in the current process. While doing this, it preserves the alignment specified in the **o_algntext** and **o_algndata** fields of the module's **<aouthdr.h>** header. Once the base module is made addressable, the next step is to determine what, if any, dependency modules are required.

Dependency modules are specified in the loader section of the base module. The **l_impoff** field of the loader section header contains the offset to the start of the loader import file ID strings. The number of import file ids is specified in the **l_nimpid** field of the loader section header. The loader insures that the current version of each dependency module is made addressable. (Current version means the current version in the file system.)

When loading dependency modules, the loader must maintain the alignment specified in the **<aouthdr.h>** header. For each dependency, the Library Search Path information is used to determine the actual path to the file in a file system. (Refer to page 5−8 for a description of the Library Search Path.) After making a dependency module addressable, the dependencies of the dependency module must be made addressable in the same manner. Repeat until all dependency modules have been made addressable.

If the loader is unable to find any of the dependency modules, an error is returned to the calling process. (See the **exec** in the API for more information.)

5.2.2 Import/Export Symbols

Import/Export symbols tie the loaded modules in a process image together. Application programmers must be aware of what symbols are needed by a module to insure that the module imports them. The programmer must also decide what symbols should be made available to other modules and export them.

The list of symbols examined by the system's loader is in the symbol table of the module's loader section. The **l_smtype** field of a loader symbol entry identifies a symbol as an Import and/or Export. For Import symbols, the **l_ifile** field of the loader symbol entry identifies the import file which should export the specified symbol. The **l_ifile** field is an index into the list of import files required by the module. The list is in the loader section of the file and is the same list used to locate dependency modules for the given module. When the **l_ifile** field of an Import symbol is NULL, the symbol is marked for deferred resolution. (Refer to page 5−7, Symbol Resolution, for information on deferred resolution.) A symbol marked for Export is available for use by other modules. The **l_ifile** field is not used for Export symbols because the name of the exporting module is the current module.

It is possible for a symbol to be marked both Import and Export. In this case, the module will import the symbol from the defining module and export it for use in other modules. The **l_ifile** field of the symbol table entry still identifies the importing module, and the exporting module remains the current module.

The **l_smtype** field can also be used to identify which symbol is the entry point for the given module. In a **load** system call, the loaded module's entry point is returned to the calling program. In an **exec** system call, control is passed to the entry point.

5.2.3 Symbol Resolution

Symbol resolution is the process by which one obtains the actual values of symbols imported by a module. It occurs after the base module and all dependency modules are successfully made addressable. The loader will only perform symbol resolution for each module specified as a base module or dependency in the current load. For each module, the loader will examine the list of symbols specified in the symbol table of the module's loader section. The **l_nsyms** field of the loader section header contains the number of symbol entries in the loader section. The loader searches for symbols of type L_IMPORT. (The **l_smtype** field of a loader symbol entry gives the type.) For each Import symbol, the import file ID (**l_ifile**) determines which module in the current load should export the symbol.

The symbol table of the module that exports the symbol is then searched to insure that it does export the symbol. If the symbol is not found, an error is returned to the calling process. Once the exporting symbol is found, its address can be determined, and this is used to determine the actual value of the symbol in the Importing module. A symbol which has an import file ID of 0 is marked for deferred resolution and no symbol resolution is done at load time. Symbol resolution repeats for each module in the current load.

Symbol resolution is successful if all imported symbols have been resolved.

The dependency /**unix** is a special case that represents the kernel name space. It includes the system calls contained in the operating system kernel as well as any symbols exported by the kernel. Modules that reference system calls will have a dependency on /**unix**. but this dependency does not mean that a new version of /**unix** should be made addressable.

5.2.4 Relocation

Relocation is the process by which data in a module that is dependent on either its loaded location or on the address of an imported symbol is updated with appropriate values. Relocation is done by processing the relocation entries in the loader section of each module in the current load. The **l_nreloc** field of the loader section header contains the number of relocation entries for the loaded module. Each relocation entry contains the following information:

l_vaddr — Virtual address within the module to relocate. By using the virtual address the actual address is calculated as follows:

l_vaddr minus **s_vaddr** (virtual address of data section for this module) plus actual location of loaded data section for this module.

l_symndx — The symbol index by which one should relocate. Note that symbol indexes 0, 1, and 2 are defined and refer to the .text, .data and .bss sections, respectively.

l_rtype — This field contains both the relocation type and relocation size. **l_rtype** & 0x00ff will provide the relocation type. ((**l_rtype** >> 8) & 0x1f) + 1 is the relocation size in bits. Note that #bits / 8 is the number of bytes affected, and the bits affected are right justified within the bytes affected. This relocation type can be any of the following:

R_POS, **R_BA**, **R_RBA**, **R_RLA**, **R_RL**, **R_TCL**, or **R_GL**
The actual address of the symbol specified in the **l_symndx** field is added to the relocation size bits at the target location specified by **l_vaddr**.

R_NEG
The actual address of the symbol specified in the **l_symndx** field is negated and added to the *relocation size* bits at the target location specified by **l_vaddr**.

R_REL, **R_TRLA**, **R_BR**, **R_RBR**, **R_TOC**, or **R_TRL**
The data relocation value for the module is subtracted from the value of the symbol specified in the **l_symndx** field. (The data relocation value is calculated as the actual location of loaded data minus the virtual address of data as specified by **s_vaddr**.) This result is added to the *relocation size* bits at the target location specified by **l_vaddr**.

R_REF,
No relocation is performed.

l_rsecnm The section number in which the target relocation address lies. It must be the data section. In a future release of the PowerOpen ABI l_rsecnm may be enhanced to support other sections than .data.

Relocation is performed on each module in the current load. If a symbol referenced in a relocation entry is marked for deferred resolution, no relocation is performed for that entry. If an error occurs during relocation, both control and an error are returned to the calling process.

After all modules in the current load have been relocated they can be used by the calling process. The entry point (as specified in the base module) is determined as described under Import/Export Symbols on page 5−7. In a **load** system call, the entry point is returned to the caller. In an **exec** system call, control is passed to the entry point.

5.2.5 Deferred Symbol Resolution

Symbols may be marked for deferred resolution. Deferred resolution means the symbols can remain unresolved when the module is initially loaded. The process using this module is responsible for ensuring that the symbol is resolved before the symbol is accessed. If the module was loaded by an **exec** system call as a base module or dependency, or if it was loaded with the **load** system call and the **L_NOAUTODEFER** flag is not set, then the symbol will be resolved at the earliest opportunity when any module is loaded that has an exported symbol matching the deferred import symbol. If a module is loaded with the **load** function and **L_NOAUTODEFER** is set, then the symbol will not be resolved until an explicit call to the **loadbind** system call is made.

5.2.6 Library Search Path

Before modules are loaded, they must be located in a filesystem. A Library Search Path is a colon-separated list of strings used by the loader to locate load modules in a file system.

Each loadable module contains a list of import file IDs. This list is contained in the loader section of the module's object file. Each entry in the list of import file IDs is comprised of the following three separate parts:

Path The specified load path

Base The base module name

Member The member within the base module, used for archive files only

The first entry in a modules import file ID list is treated as a special case. The path part of this entry is used as the default library search path for the module.

At the beginning of the **exec** or **load** process, or at the time of a load system call, the loader will determine the library search path to be used for the duration of the load.

5.2.6.1 The exec System Call

When a module is loaded by **exec**, the loader obtains the current value of the LIBPATH environment variable. If this variable is set, then the value of the LIBPATH environment variable will be used as the library search path. If the LIBPATH environment variable is not set, then the loader will obtain the path information contained in the exec(base) module. The library search path will be set to this string. If the exec'ed module is root then the LIBPATH environment variable is ignored.

5.2.6.2 The load System Call

The load system call takes a parameter that allows the caller to specify a library path to be used during the load. If this library path parameter is set, then this string is used to search for the specified load module. The library search path is set to this string.

If the library path parameter is not set, and the LIBPATH environment variable is set, then the LIBPATH environment variable is used to search for the specified load module. The library search path is set to the value of the LIBPATH environment variable.

If neither the libpath parameter, or the LIBPATH environment variable is set, then the loader will look for the load module in the current directory. If the load module is found in the current directory, then the loader will obtain the path information contained in the load module. The library search path will be set to this string.

The library search path is used for the duration of the load. When a module is loaded, the library search path is prepended to the path string contained in the module itself. The resulting string is then used to search for the dependencies of the module. This process is repeated for each dependency module until all dependency modules are located.

5.3 Loader Interfaces

There are five system call interfaces to the system's loader. They are as follows:

exec Destroys the current process image and loads the specified program image. This call will perform dependency checking and load all required modules for the new process image. After the new process is loaded, control is passed to the entry point of the loaded module.

load Loads the specified module and its dependencies into the current process image. The load system call returns the entry point of the loaded module.

unload Unloads the specified module and all its dependencies from the current process image.

loadbind Used to cause the system's loader to resolve any deferred imported symbols in one specified module to exported symbols in another specified module. This service is provided to support the specification of the exporting module at run time instead of at bind time.

loadquery Provides an interface to query the system's loader about modules loaded into the current process image and error status.