Dynamic Linking and Loading

$Revision: 1.1 $
$Date: 1999/01/02 09:38:11 $

Dynamic linking defers much of the linking process until a program starts running. It provides a variety of benefits that are hard to get otherwise:

- Dynamically linked shared libraries are easier to create than static linked shared libraries.
- Dynamically linked shared libraries are easier to update than static linked shared libraries.
- The semantics of dynamically linked shared libraries can be much closer to those of unshared libraries.
- Dynamic linking permits a program to load and unload routines at runtine, a facility that can otherwise be very difficult to provide. There are a few disadvantages, of course. The runtime performance costs of dynamic linking are substantial compared to those of static linking, since every symbol has to be looked up in a symbol table. (Microsoft DLLs mitigate this cost somewhat, as we describe below.) Dynamic libraries are also somewhat larger than static libraries, since the dynamic ones have to include symbol tables.

Beyond issues of call compatibility, a chronic source of problems is changes in library semantics. Since dynamic shared libraries are so easy to update compared to unshared or static shared libraries, it's easy to change libraries that are in use by existing programs, which means that the behavior of those programs changes even though "nothing has changed". This is a frequent source of problems on Microsoft Windows, where programs use a lot of shared libraries, libraries go through a lot of versions, and library version control is not very sophisticated. Most programs ship with copies of all of the libraries they use, and installers often will inadvertently install an older version of a shared library on top of a newer one, breaking programs that are expecting features found in the newer one. Well-behaved applications pop up a warning before installing an older library over a newer one, but even so, programs that depend on semantics of older libraries have been known to break when newer versions replace the older ones.

ELF dynamic linking

Sun Microsystems' SunOS introduced dynamic shared libraries to UNIX in the late 1980s. UNIX System V Release 4, which Sun co-developed, introduced the ELF object format and adapted the Sun scheme to ELF. ELF was clearly an improvement over the previous object formats, and by the late 1990s it had become the standard for UNIX and UNIX like systems including Linux and BSD derivatives.

Contents of an ELF file

As mentioned in Chapter 3, an ELF file can be viewed as a set of *sections*, interpreted by the linker, or a set of *segments*, interpreted by the program loader. ELF programs and shared libraries have the same general structure, but with different sets of segments and sections.

ELF shared libraries can be loaded at any address, so they invariably use position independent code (PIC) so that the text pages of the file need not be relocated and can be shared among multiple processes. As described in Chapter 8, ELF linkers support PIC code with a Global Offset Table (GOT) in each shared library that contains pointers to all of the static data referenced in the program, Figure 1. The dynamic linker resolves and relocates all of the pointers in the GOT. This might seem to be a performance issue but in practice the GOT is small; a commonly used version of the standard C library has only 180 entries in the GOT for over 350K of code.

Since the GOT is in the same loadable ELF file as the code that references it, and the relative addresses within a file don't change regardless of where the program is loaded, the code can locate the GOT with a relative address, load the address of the GOT into a register, and then load pointers from the GOT whenever it needs to address static data. A library need not have a GOT if it references no static data, but in practice all libraries do.

To support dynamic linking, each ELF shared libary and each executable that uses shared libraries has a Procedure Linkage Table (PLT). The PLT adds a level of indirection for function calls analogous to that provided by the GOT for data. The PLT also permits "lazy evaluation", that is, not resolving procedure addresses until they're called for the first time. Since the PLT tends to have a lot more entries than the GOT (over 600 in the C library mentioned above), and most of the routines will never be called in any given program, that can both speed startup and save considerable time overall.

Figure 1: PLT and GOT

picture of program with PLT
picture of library with PLT and GOT

We discuss the details of the PLT below.

An ELF dynamically linked file contains all of the linker information that the runtime linker will need to relocate the file and resolve any undefined symbols. The `.dynsym` section, the dynamic symbol table, contains all of the file's imported and exported symbols. The `.dynstr` and `.hash` sections contain the name strings for the symbol, and a hash table the linker can use to look up symbols quickly.

The final extra piece of an ELF dynamically linked file is the `DYNAMIC` segment (also marked as the `.dynamic` section) which runtime dynamic linker uses to find the information about the file the linker needs. It's loaded as part of the data segment, but is pointed to from the ELF file header so the runtime dynamic linker can find it. The DYNAMIC section is a list of tagged values and pointers. Some entry types occur just in programs, some just in libraries, some in both.

- NEEDED: the name of a library this file needs. (Always in programs, sometimes in libraries when one library is dependend on another, can occur more than once.)

- SONAME: "shared object name", the name of the file the linker uses. (Libraries.)
- SYMTAB, STRTAB, HASH, SYMENT, STRSZ,: point to the symbol table, associated string and hash tables, size of a symbol table entry, size of string table. (Both.)
- PLTGOT: points to the GOT, or on some architectures to the PLT (Both.)
- REL, RELSZ, and RELENT or RELA, RELASZ, and RELAENT: pointer to, number of, and size of relocation entries. REL entries don't contain addends, RELA entries do. (Both.)
- JMPREL, PLTRELSZ, and PLTREL: pointer to, size, and format (REL or RELA) of relocation table for data referred to by the PLT. (Both.)
- INIT and FINI: pointer to initializer and finalizer routines to be called at program startup and finish. (Optional but usual in both.)
- A few other obscure types not often used. An entire ELF shared library might look like Figure 2. First come the read-only parts, including the symbol table, PLT, text, and read-only data, then the read-write parts including regular data, GOT, and the dynamic section. The bss logically follows the last read-write section, but as always isn't present in the file.

Figure 2: An ELF shared library

(Lots of pointer arrows here)

read-only pages:
.hash
.dynsym
.dynstr
.plt
.text

.rodata

read-write pages:
.data
.got
.dynamic

.bss

An ELF program looks much the same, but in the read-only segment has init and fini routines, and an INTERP section in the
to specify the name of the dynamic linker (usually `ld.so`). The data segment has no GOT, since programs aren't relocated at runtime.

Loading a dynamically linked program

Loading a dynamically linked ELF program is a lengthy but straightforward process. When the operating system runs the program, it maps in the file's pages as normal, but notes that there's an INTERPRETER section in the executable. The specified interpreter is the dynamic loader, ld.so, which is itself in ELF shared library format, so the system maps that into a convenient part of the address space as well and starts the loader, passing on the stack some arguments describing the program to run including pointers to base and entry point of the loaded executable.

At this point, bootstrap code at the beginning of the loader finds the loader's GOT, the first entry in which points to the DYNAMIC segment in the loader file. From the dynamic segment, the loader can find its own relocation entries and relocate pointers in its own data segment. It then initializes a chain of symbol tables with pointers to the program's symbol table and the loader's own symbol table.

Once it's done that, it finds the names of the libraries required by the program. For each library, the loader finds the library's ELF shared library file (using a file containing a cache of full pathnames), maps the library's text and data segments in, finds the library's dynamic segment, adds the library's symbol table to the chain of symbol tables, and if the library requires further libraries not already used, adds any new libraries to the list.

When this process terminates, all of the libraries have been mapped in, and the loader has a logical global symbol table consisting of the union of all of the symbol tables of the program and the mapped library. Now the loader revisits each library and handles the library's relocation entries, filling in the library's GOT and performing any relocations needed in the library's data segment. If a library has an `.init` section, the loader calls it to do library-specific initializations, such as C++ static constructors, and any `.fini` section is noted to be run at exit time. When this pass is done, all of the libraries are fully loaded and ready to execute, and the loader calls the program's entry point to start the program.

Lazy procedure linkage with the PLT

Programs that use shared libraries generally contain calls to a lot of functions, but any single run of the program rarely executes all of the calls, since many of them are in error routines or other parts of the program that aren't used. Furthermore, each a shared library also contains a lot of calls, even fewer of which will be executed in a given program run since many of them are in routines that the program never calls either directly or indirectly. To speed program startup, dynamically linked ELF programs use lazy binding of procedure addresses. That is, the address of a procedure isn't bound until the first time the procedure is called.

ELF supports lazy binding via the Procedure Linkage Table, or PLT. Each dynamically bound program and shared library has a PLT, with the PLT containing an entry for each global routine called from the program or library, Figure 3. Note that the PLT in PIC code is itself PIC, so it can be part of the read-only text segment.

Figure 3: PLT structure

Special first entry

```
PLT0:  pushl  GOT+4
  jmp *GOT+8
```

Regular entries, non-PIC code:

```
PLTn:  jmp *GOT+m
  push #reloc_offset
  jmp PLT0
```

Regular entries, PIC code:

```
PLTn:  jmp *GOT+m(%ebx)
  push #reloc_offset
  jmp PLT0
```

All calls within the program or library to a particular routine are adjusted by the linker to be calls to the routine's entry in the PLT. The first time the program or library calls a routine, the PLT entry calls the runtime linker to resolve the actual address of the routine. After that, the PLT entry jumps directly to the actual address, so after the first call, the cost of using the PLT is a single extra jump.

The first entry in the PLT, which we call PLT0, is special code to call the dynamic linker. At load time, the dynamic linker automatically places two values in the GOT, a code that identifying the library GOT+4, and the address of the dynamic linker's symbol resolution routine at GOT+8.

The rest of the entries in the PLT, which we call PLTn, with n running from 1 to the number of PLT entries, each start with an indirect jump through a GOT entry. Each PLT entry has a corresponding GOT entry which is initially set to point to the push instruction in the PLT entry after the initial jmp. (In a PIC file this will require a loadtime relocation, but not an expensive symbol lookup.) Following the jump is a push instruction which pushes a relocation offset. The offset is an offset in the file's relocation table of a special relocation entry of type `R_386_JMP_SLOT`. The relocation entry's symbol reference points to the symbol in the files, symbol table, and its address points to the GOT entry.

This compact but rather baroque arragement means that the first time the program or library calls a PLT entry, the first jump in the PLT entry in effect does nothing, since the GOT entry through which it jumps points back into the PLT entry. Then the push instruction pushes the offset value which indirectly identifies both the symbol to resolve and the GOT entry into which to resolve it, and jumps to PLT0. The instructions in PLT0 push another code that identifies which program or library it is, and then jumps into stub code in the dynamic linker.

At this point the stub code saves all the registers and calls an internal routine to do the resolution. It gets the symbol name, looks up the symbol value using the concatenated runtime symbol table, and stores the routine's address into the GOT entry. Then the stub code restores the registers pops the two words that the PLT pushed, and jumps off to the routine. Now the GOT entry has been updated, so subsequent calls to that PLT entry jump off to the routine itself.

Other peculiarities of dynamic linking

The ELF linker and dynamic linker have a lot of obscure code to handle special cases and try and keep the runtime semantics as similar as possible to whose of unshared libraries.

Static initializations

If a program has an external reference to a global variable defined in a shared library, the linker has to create a copy of the variable in the program, since program data addresses have to be bound at link time, Figure 4. This poses no problem for the code in the shared library, since the code refers to the variable via a GOT pointer which the dynamic linker can fix up, but there is a problem if the library initializes the variable. To deal with this problem, the linker puts an entry in the program's relocation table (which otherwise just contains `R_386_JMP_SLOT` entries) of type `R_386_COPY` that points to the place in the program where the copy of the variable is defined, and tells the dynamic linker to copy the initial value of that word of data from the shared library.

Figure 4: Global data initialization

Main program:

```
extern int token;
```

Routine in shared library:

```
int token = 42;
```

Library versions

Dynamic libraries are generally named with major and minor versions numbers, like `libc.so.1.1` but programs should be bound only to major version numbers like `libc.so.1` since minor versions are supposed to be upward compatible.

To keep program loading reasonably fast, the system manager maintains a cache file containing the full pathname most recent version of each library, which is updated by a configuration program whenever a new library is installed.

To support this design, each dynamically linked library can have a "true name" called the *SONAME* assigned at library creation time. For example, the library called `libc.so.1.1` would have a SONAME of `libc.so.1`. (The SONAME defaults to the library's name.) When the linker builds a program that uses shared libraries, it lists the SONAMEs of the libraries it used rather than the actual names of the libraries. The cache creation program scans all of the directories that contain shared libraries, finds all of the shared libraries, extracts the SONAME from each one, and where there are multiple libraries with the same SONAME, discards all but the highest version number. Then it writes the cache file with SONAMEs and full pathnames so at runtime the dynamic linker can quickly find the current version of each library.

Library interposition

Sometimes it's useful to force the dynamic linker to bind to the "wrong" library. Users can define a list of directories in an environment variable `LD_LIBRARY_PATH` that the runtime linker searches before looking for libraries in the standard place. This means that one can build a new version of a library, put it in the `LD_LIBRARY_PATH` and use it with existing linked programs either to test the new library, or equally well to instrument the behavior of the program.

Dynamic loading at runtime

Although the ELF dynamic linker is usually called implcitly at program load time and from PLT entries, programs can also call it explicitly using `dlopen()` to load a shared library and `dlsym()` to find the address of a symbol, usually a procedure to call. Those two routines are actually simple wrappers that call back into the dynamic linker. When the dynamic linker loads a library via `dlopen()`, it does the same relocation and symbol resolution it does on any other library, so the dynamically loaded program can without any special arrangements call back to routines already loaded and refer to global data in the running program.

This permits users to add extra functionality to programs without access to the source code of the programs and without even having to stop and restart the programs (useful when the program is something like a web server.) Mainframe operating systems have provided access to "exit routines" like this since at least the early 1960s, albeit without such a convenient interface, but it's long been a way to add great flexibility to packaged applications. It also provides a way for programs to extend themselves; there's no reason a program couldn't write a routine in C or C++, run the compiler and linker to create a shared library, then dynamically load and run the new code. (Mainframe sort programs have linked and loaded custom inner loop code for each sort job for decades.)

Microsoft Dynamic Link Libraries

Microsoft Windows also provides shared libraries, called dynamic-link libraries or DLLs in a fashion similar to but somewhat simpler than ELF shared libraries. The design of DLLs changed substantially between the 16 bit Windows 3.1 and the 32 bit Windows NT and 95. This discussion addresses only the more modern Win32 libraries. Although the design of DLLs would make it possible to export data addresses using a GOT-like scheme, in practice DLLs only export procedure addresses using a PLT-like scheme.

In Windows, both programs and DLLs are ECOFF format PE (portable executable) files intended to be memory mapped into a process. Unlike Windows 3.1, where all applications shared a single address space, Win32 gives each application its own address space and executables and libraries are mapped into each address space where they are used. For read-only code this doesn't make any practical difference, but for data it means that each application using a DLL gets its own copy of the DLL's data. (That's a slight oversimplification, since ECOFF files can mark some sections as shared data with a single copy shared among all applications that use the file, but most data is unshared.)

Loading a Windows executable and DLLs is similar to loading a dynamically linked ELF program, although in the Windows case the dynamic linker is part of the kernel. First the kernel maps in the executable file, guided by section info in the COFF headers. Then it maps in all of the DLLs, again guided by the COFF headers in each DLL. PE files can contain relocation entries. An Executable generally won't contain them and so have to be mapped at the address for which it was linked. DLLs all do contain relocation entries, and are relocated when they're mapped in if the address space for which they were linked isn't available. (Microsoft calls runtime relocation *rebasing*.)

All PE files, both executables and DLLs, have an entry point, and the loader calls a DLL's entry point when the DLL is loaded, when the DLL is unloaded, and each time a process thread attaches to or detaches from the DLL. (The loader passes an argument to say why it's making each call.) This provides a hook for static initializers and destructors analogous to the ELF `.init` and `.fini` sections.

ECOFF supports shared libraries with two special sections of the file, `.edata`, for exported data, that lists the symbols exported from a file, and `.idata`, that lists the symbols imported into a file. Program files generally have only an `.idata` section, while DLLs always have an `.edata` and may have a `.idata` if they use other DLLs. Symbols can be exported either by symbol name, or by "ordinal", a small integer that gives the index of the symbol in the export address table. Linking by ordinals is slightly more efficient since it avoids a symbol lookup, but considerably more error prone since it's up to the person who builds a DLL to ensure that ordinals stay the same from one library version to another. In practice ordinals are usually used to call system services that rarely change, and names for everything else.

The `.edata` section contains an export directory table that describes the rest of the section, followed by the tables that define the exported symbols, Figure 5.

Figure 5: Structure of .edata section

export directory pointing to:
export address table
ordinal table
name pointer table
name strings

The export address table contains the RVA (relative virtual address, relative to the base of the ECOFF file) of the symbol. If the RVA points back into the `.edata` section, it's a "forwarder" reference, and the value pointed to is a string naming the symbol to use to satisfy the reference, probably defined in a different DLL. The ordinal and name pointer table are parallel, with each entry in the name pointer table being the RVA of the name string for the symbol, and the ordinal being the index in the export address table. (Ordinals need not be zero-based; the ordinal base to subtract from ordinal values to get the index in the export address table is stored in the export directory and is most often 1.) Exported symbols need not all have names, although they invariably all do. The symbols in the name pointer table are in alphabetical order to permit the loader to use a binary search.

The `.idata` section does the converse of what the `.edata` section does, by mapping symbols or ordinals back into virtual addresses. The section consists of a null-terminated array if import directory tables, one per DLL from which symbols are imported, followed by an import lookup table per DLL, followed by a name table with hints, Figure 5.

Figure 5: Structure of .idata section

array of import directory tables, with lotsa arrows
each has import lookup table RVA, time/date stamp, forwarder chain (unused?), DLL name, import address RVA table
NULL
import table, entries with high bit flag (table per DLL)
hint/name table

For each imported DLL, there is an array of import addresses, typically in the program's text segment, into which the program loader places the resolved addresses. The import lookup table that identifies the symbols to import, with the entries in the import lookup table being parallel to those in the import address table. The lookup table consists of 32 bit entries. If the high bit of an entry is set, the low 31 bits are the ordinal of the symbol to import, otherwise the entry is the RVA of an entry in the hint/name table. Each hint/name entry consists of a four-byte hint that guesses the index of the symbol in the DLL's export name pointer table, followed by the null terminated symbol name. The program loader uses the hint to probe the export table, and if the symbol name matches, it uses that symbol, otherwise it binary searches the entire export table for the name. (If the DLL hasn't changed, or at least its list of exported symbols hasn't changed, since the program that uses the DLL was linked, the guess will be right.)

Unlike ELF imported symbols, the values of symbols imported via `.idata` are only placed in the import address table, not fixed up anywhere else in the importing file. For code addresses, this makes little difference. When the linker builds an executable or DLL, it links into the text section a table of misnamed "thunks" that are indirect jumps through the entries in the import address table, and uses the addresses of the thunks as the address of the imported routine, which is transparent to the programmer. (The thunks as well as most of the data in the `.idata` section actually come from a stub library created at the same time as the DLL.) In recent versions of Microsoft's C and C++ compiler, if the programmer knows that a routine will be called in a DLL, the routine can be declared "dllimport", and the compiler will emit an indirect call to the address table entry, avoiding the indirect jump. For data addresses, the situation is more problematical, since it's harder to hide the extra level of indirection required to address a symbol in another executable. Traditionally, programmers just bit the bullet and explicitly declared imported variables to be pointers to the real values and explicitly dereferencd the pointers. Recent versions of Microsoft's C and C++ compiler also let you declare global data to be "dllimport" and the compiler will emit the extra pointer deferences, much like ELF code that references data via pointers in the GOT.

Recent versions of Windows compilers have added delay loaded imports to permit lazy symbol binding for procedures, somewhat like the ELF PLT. A delay-loaded DLL has a structure similar to the `.idata` import directory table, but not in the *.idata* section so the program loader doesn't handle it automatically. The entries in the import address table initially all point to a helper routine that goes and loads the DLL and replaces the contents of the address table with the actual addresses. The delay-loaded directory table has a place to store the original contents of the import address table so the values can be put back if the DLL is later unloaded. Microsoft provides a standard helper routine, but its interfaces are documented and programmers can write their own versions if need be.

Windows also permits programs to load and unload DLLs explicitly using `LoadLibrary` and `FreeLibrary`, and to find addresses of symbols using `GetProcAddress`.

OSF/1 pseudo-static shared libraries

OSF/1, the ill-fated UNIX variant from the Open Software Foundation, used a shared library scheme intermediate between static and dynamic linking. Its authors noted that static linking is a lot faster than dynamic since less relocation is needed, and that libraries are updated infrequently enough that system managers are willing to endure some pain when they update shared libraries, although not the agony of relinking every executable program in the entire system.

So OSF/1 took the approach of maintaining a global symbol table visible to all processes, and loaded all the shared libraries into a sharable address space at system boot time. This assigned all of the libraries addresses that wouldn't change while the system was running. Each time a program started, if it used shared libraries, it would map in the shared libraries and symbol table and resolve undefined references in the executable using the global symbol table. No load-time relocation was ever required since programs were all linked to load in a part of the address space that was guaranteed to be available in each process, and the libraries had already been relocated when they were loaded at boot time.

When one of the shared libraries changed, the system just had to be rebooted normally, at which point the system loaded the new libraries and created a new symbol table for executables to use.

This scheme was clever, but it wasn't very satisfactory. For one thing, processing symbol lookups is considerably slower than processing relocation entries, so avoiding relocation wasn't that much of a performance advantage. For another, dynamic linking provides the ability to load and run a library at runtime, and the OSF/1 scheme didn't provide for that.

Exercises

In ELF shared libraries, libraries are often linked so that calls from one routine to another within the shared library go through the PLT and have their addresses bound at runtime. Is this useful? Why or why not?

Imagine that a program calls a library routine `plugh()` that is found in a shared library, and the programmer builds a dynamically linked program that uses that library. Later, the system manager notices that `plugh` is a silly name for a routine and installs a new version of the library that calls the routine `xsazq` instead. What happens when the next time the programmer runs the program?

If the runtime environment variable `LD_BIND_NOW` is set, the ELF dynamic loader binds all of the program's PLT entries at load time. What would happen in the situtation in the previous problem if `LD_BIND_NOW` were set?

Microsoft implemented lazy procedure binding without operating system assistance by adding some extra cleverness in the linker and using the existing facilities in the operating system. How hard would it be to provide transparent access to shared data, avoiding the extra level of pointers that the current scheme uses?