本科毕业设计（论文）英文翻译

学 生 姓 名： 袁野

学　　号： 04212069

班　　级： 网络 2102

专　　业： 网络工程

院　（系）： 计算机学院

指 导 教 师： 王亚刚

二零二五年五月

一、英文原文

起止页码： 247~272

出版日期（期刊号）： 1999 年 10 月 25 日

刊物名称（出版单位）：Linkers & Loaders

Chapter 10
Dynamic Linking and Loading

Dynamic linking defers much of the linking process until a program startsrunning. 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 a large part of the linking process has to be redone every time a program runs. Every dynamically linked symbol used in a program has to belooked up in a symbol table and resolved. (Windows DLLs mitigate this cost somewhat, as we describe below.) Dynamic libraries are also 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 pro- grams 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 inad- vertently 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 li- brary 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 sys- tems 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 struc- ture, 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 re- locates all of the pointers in the GOT. This can be a performance issue but in practice the GOT is small except in very large libraries; 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 re- solving 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.

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 run- time 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 load- ed 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 pro- grams, 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 us- es. (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.

Loading a dynamically linked program

Loading a dynamically linked ELF program is a lengthy but straightfor- ward process.

Starting the dynamic linker

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 ex- ecutable. The specified interpreter is the dynamic linker, ld.so, which is it- self in ELF shared library format. Rather than starting the program, the system maps the dynamic linker into a convenient part of the address space as well and starts ld.so, passing on the stack an *auxiliary vector* of information needed by the linker. The vector includes:

• AT_PHDR, AT_PHENT, and AT_PHNUM: The address of the program header for the program file, the size of each entry in the header, and the number of entries. This structure describes the seg- ments in the loaded file. If the system hasn't mapped the program into memory, there may instead be a AT_EXECFD entry that con- tains the file descriptor on which the program file is open.

• AT_ENTRY: starting address of the program, to which the dynam- ic linker jumps after it has finished initialization.

• AT_BASE: The address at which the dynamic linker was loaded At this point, bootstrap code at the beginning of ld.so finds its own GOT,

the first entry in which points to the DYNAMIC segment in the ld.so file.

From the dynamic segment, the linker can find its own relocation entries, relocate pointers in its own data segment, and resolve code references to the routines needed to load everything else. (The Linux ld.so names all of the essential routines with names starting with _dt_ and special-case code

looks for symbols that start with the string and resolves them.)

The linker then initializes a chain of symbol tables with pointers to the program's symbol table and the linker's own symbol table. Conceptually, the program file and all of the libraries loaded into a process share a single symbol table. But rather than build a merged symbol table at runtime, the linker keeps a linked list of the symbol tables in each file. each file con- tains a hash table to speed symbol lookup, with a set of hash headers and a hash chain for each header. The linker can search for a symbol quickly by computing the symbol's hash value once, then running through apprpriate hash chain in each of the symbol tables in the list.

Finding the libraries

Once the linker's own initializations are done, it finds the names of the li- braries required by the program. The program's program header has a pointer to the "dynamic" segment in the file that contains dynamic linking information. That segment contains a pointer, DT_STRTAB, to the file's string table, and entries DT_NEEDED each of which contains the offset in the string table of the name of a required library.

For each library, the linker finds the library's ELF shared library file, which is in itself a fairly complex process. The library name in a DT_NEEDED entry is something like *libXt.so.6* (the Xt toolkit, version 6.) The library file might in in any of several library directories, and might not even have the same file name. On my system, the actual name of that library is /usr/X11R6/lib/libXt.so.6.0, with the ‘‘.0’’ at the end being a mi- nor version number.

The linker looks in these places to find the library:

• If the dynamic segment contains an entry called DT_RPATH, it's a colon-separated list of directories to search for libraries. This entry is added by a command line switch or environment variable to the regular (not dynamic) linker at the time a program is linked. It's mostly used for subsystems like databases that load a collection of programs and supporting libraries into a single directory.

• If there's an environment symbol LD_LIBRARY_PATH, it's treat- ed as a colon-separated list of directories in which the linker looks for the library. This lets a developer build a new version of a li- brary, put it in the LD_LIBRARY_PATH and use it with existing linked programs either to test the new library, or equally well to in- strument the behavior of the program. (It skips this step if the pro- gram is set-uid, for security reasons.)

• The linker looks in the library cache file /etc/ld.so.conf which contains a list of library names and paths. If the library name is present, it uses the corresponding path. This is the usual

way that most libraries are found. (The file name at the end of the path need not be exactly the same as the library name, see the sec- tion on library versions, below.)

• If all else fails, it looks in the default directory /usr/lib, and if the library's still not found, displays an error message and exits.

Once it's found the file containing the library, the dynamic linker opens the file, and reads the ELF header to find the program header which in turn points to the file's segments including the dynamic segment. The linker allocates space for the library's text and data segments and maps them in, along with zeroed pages for the bss. From the library's dynamic segment, it adds the library's symbol table to the chain of symbol tables, and if the library requires further libraries not already loaded, adds any new libraries to the list to be loaded.

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.

Shared library initialization

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. Load-time relocations on an x86 include:

• R_386_GLOB_DAT, used to initialize a GOT entry to the address of a symbol defined in another library.

• R_386_32, a non-GOT reference to a symbol defined in another li- brary, generally a pointer in static data.

• R_386_RELATIVE, for relocatable data references, typically a pointer to a string or other locally defined static data.

• R_386_JMP_SLOT, used to initialize GOT entries for the PLT, de- scribed later.

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. (It doesn't do the init for the main program, since that's handled in the program's own startup code.) 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.

二、中文译文

第 10 章　动态链接和加载

动态链接将大部分链接过程推迟到程序运行时。它提供了许多通过其他方式难以获得的好处：

- 动态链接共享库比静态链接共享库更容易创建。
- 动态链接共享库比静态链接共享库更容易更新。

- 动态链接共享库的语义可以更接近非共享库的语义。
- 动态链接允许程序在运行时加载和卸载例程，这是一种否则很难提供的功能。

当然，也有一些缺点。动态链接的运行时性能成本相对于静态链接来说是巨大的，因为每次程序运行都需要重新进行大部分链接过程。程序中使用的每个动态链接符号都必须在符号表中查找并解析。（Windows DLLs 在一定程度上缓解了这一成本，正如我们将在下面描述的那样。）动态库也比静态库大，因为动态库必须包含符号表。

除了调用兼容性问题，一个长期存在的问题是库语义的改变。由于动态共享库比非共享或静态共享库更容易更新，因此很容易更改正在被现有程序使用的库，这意味着即使“什么都没有改变”，这些程序的行为也会改变。这在 Microsoft Windows 上是一个常见的问题来源，因为程序使用大量共享库，库经历许多版本，并且库版本控制不是很复杂。大多数程序都随附其使用的所有库的副本，并且安装程序通常会无意中将旧版本的共享库安装到新版本之上，从而破坏期望新版本中功能的程序。

行为良好的应用程序在安装旧库覆盖新库之前会弹出一个警告，但即便如此，依赖旧库语义的程序也已知会在新版本替换旧版本时出现问题。

ELF 动态链接

Sun Microsystems 的 SunOS 在 20 世纪 80 年代后期将动态共享库引入 UNIX。Sun 共同开发的 UNIX System V Release 4 引入了 ELF 对象格式，并将 Sun 方案应用于 ELF。ELF 显然是对以前对象格式的改进，到 20 世纪 90 年代后期，它已成为 UNIX 和类 UNIX 系统（包括 Linux 和 BSD 派生系统）的标准。

ELF 文件的内容

正如第三章所述，一个 ELF 文件可以被看作是一组由链接器解释的节，或一组由程序加载器解释的段。ELF 程序和共享库具有相同的通用结构，但具有不同的段和节集。

ELF 共享库可以在任何地址加载，因此它们总是使用位置无关代码 (PIC)，这样文件的文本页就不需要重新定位，并且可以在多个进程之间共享。正如第八章所述，ELF 链接器通过每个共享库中的全局偏移表 (GOT) 支持 PIC 代码，该表包含指向程序中所有静态数据引用的指针（图 1）。动态链接器解析并重新定位 GOT 中的所有指针。这可能会是一个性能问题，但实际上除了非常大的库外，GOT 都很小；一个常用版本的标准 C 库有超过 350K 的代码，但只有 180 个 GOT 条目。

由于 GOT 与引用它的代码位于同一个可加载 ELF 文件中，并且文件内的相对地址无论程序加载到何处都不会改变，因此代码可以通过相对地址找到 GOT，将 GOT 的地址加载到寄存器中，然后每当需要寻址静态数据时就从 GOT 中加载指针。如果一个库不引用任何静态数据，它就不需要 GOT，但实际上所有库都这样做。

为了支持动态链接，每个 ELF 共享库和每个使用共享库的可执行文件都有一个过程

链接表 (PLT)。PLT 为函数调用添加了一个间接层，类似于 GOT 为数据提供的那样。PLT 还允许“延迟评估”，即在第一次调用过程之前不解析过程地址。由于 PLT 条目往往比 GOT 多得多（上述 C 库中超过 600 个），并且在任何给定程序中，大多数例程都不会被调用，因此这既可以加快启动速度，又可以节省大量总体时间。

我们将在下面讨论 PLT 的细节。一个 ELF 动态链接文件包含运行时链接器重新定位文件和解析任何未定义符号所需的所有链接器信息。.dynsym 节（动态符号表）包含文件的所有导入和导出符号。.dynstr 和 .hash 节包含符号的名称字符串，以及运行时链接器可以用来快速查找符号的哈希表。

ELF 动态链接文件的最后一个额外部分是 DYNAMIC 段（也标记为 .dynamic 节），运行时动态链接器使用它来查找链接器所需的文件信息。它作为数据段的一部分加载，但从 ELF 文件头指向它，以便运行时动态链接器可以找到它。DYNAMIC 节是标记值和指针的列表。有些条目类型只出现在程序中，有些只出现在库中，有些则两者兼有。

- NEEDED：此文件所需的库名称。（程序中总是存在，有时一个库依赖另一个库时也会出现在库中，可以出现多次。）
- SONAME：“共享对象名称”，链接器使用的文件名称。（库。）
- SYMTAB, STRTAB, HASH, SYMENT, STRSZ：指向符号表、相关字符串和哈希表、符号表条目的大小、字符串表的大小。（两者。）
- PLTGOT：指向 GOT，或者在某些架构上指向 PLT。（两者。）
- REL, RELSZ, and RELENT 或 RELA, RELASZ, and RELAENT：指向重定位条目的指针、数量和大小。REL 条目不包含附加项，RELA 条目包含。（两者。）
- JMPREL, PLTRELSZ, and PLTREL：指向 PLT 引用数据的重定位表的指针、大小和格式（REL 或 RELA）。（两者。）
- INIT and FINI：指向在程序启动和结束时调用的初始化和终结例程的指针。（可选但通常两者都有。）
- 其他一些不常用但较不常见的类型。

一个 ELF 程序看起来大致相同，但在只读段中包含 init 和 fini 例程，并且文件前端附近有一个 INTERP 节，用于指定动态链接器的名称（通常是 ld.so）。数据段没有 GOT，因为程序文件在运行时不会重新定位。

加载动态链接程序

加载动态链接的 ELF 程序是一个漫长但直接的过程。

启动动态链接器

当操作系统运行程序时，它会像往常一样映射文件的页面，但会注意到可执行文件中有一个 INTERPRETER 节。指定的解释器是动态链接器 ld.so，它本身是 ELF 共享库格式。系统不是启动程序，而是将动态链接器也映射到地址空间的方便部分，并启动 ld.so，

在堆栈上传递一个链接器所需信息的辅助向量。该向量包括：

- AT_PHDR, AT_PHENT, and AT_PHNUM：程序文件的程序头地址、头中每个条目的大小以及条目数量。此结构描述了已加载文件中的段。如果系统尚未将程序映射到内存中，则可能改为存在一个 AT_EXECFD 条目，其中包含打开程序文件的文件描述符。
- AT_ENTRY：程序的起始地址，动态链接器在完成初始化后会跳转到该地址。
- AT_BASE：动态链接器加载的地址。

此时，ld.so 开头的引导代码会找到它自己的 GOT，其中第一个条目指向 ld.so 文件中的 DYNAMIC 段。通过动态段，链接器可以找到自己的重定位条目，重新定位其数据段中的指针，并解析加载其他所有内容所需的例程的代码引用。（Linux ld.so 将所有基本例程命名为以 dt 开头的名称，并且特殊情况代码会查找以该字符串开头的符号并解析它们。）

然后，链接器会使用指向程序符号表和链接器自身符号表的指针初始化一个符号表链。概念上，程序文件和加载到进程中的所有库共享一个单一的符号表。但链接器不是在运行时构建一个合并的符号表，而是保留每个文件中的符号表的链表。每个文件都包含一个哈希表以加快符号查找速度，其中包含一组哈希头和每个头的哈希链。链接器可以通过一次计算符号的哈希值，然后遍历列表中每个符号表中适当的哈希链来快速搜索符号。

查找库

一旦链接器自身的初始化完成，它就会找到程序所需的库名称。程序的程序头有一个指向文件中包含动态链接信息的“dynamic”段的指针。该段包含一个指向文件字符串表 DT_STRTAB 的指针，以及 DT_NEEDED 条目，每个条目都包含所需库名称在字符串表中的偏移量。

对于每个库，链接器都会找到该库的 ELF 共享库文件，这本身是一个相当复杂的过程。DT_NEEDED 条目中的库名称类似于 libXt.so.6（Xt 工具包，版本 6）。库文件可能位于几个库目录中的任何一个，甚至可能没有相同的文件名。在我的系统上，该库的实际名称是 /usr/X11R6/lib/libXt.so.6.0，末尾的“.0”是次要版本号。

链接器会在这些地方查找库：

- 如果动态段包含一个名为 DT_RPATH 的条目，它是一个冒号分隔的目录列表，用于搜索库。此条目在程序链接时通过命令行开关或环境变量添加到常规（非动态）链接器中。它主要用于数据库等子系统，这些子系统将程序集合和支持库加载到单个目录中。
- 如果存在环境变量 LD_LIBRARY_PATH，它被视为一个冒号分隔的目录列表，链接器会在其中查找库。这允许开发人员构建新版本的库，将其放入 LD_LIBRARY_PATH，并将其与现有链接程序一起使用，以测试新库，或者同样有效地测量程序的行为。（如果程序是 set-uid，出于安全原因会跳过此步骤。）
- 链接器会在库缓存文件 /etc/ld.so.conf 中查找，该文件包含库名称和路径列表。如果库名称存在，它会使用相应的路径。这是大多数库的通常查找方式。（路径末尾的文件

名不一定与库名称完全相同，请参阅下面的库版本部分。）

- 如果所有方法都失败了，它会查找默认目录 /usr/lib，如果仍然找不到库，则会显示错误消息并退出。

一旦找到包含库的文件，动态链接器就会打开该文件，并读取 ELF 头以找到程序头，该程序头又指向文件的段，包括动态段。链接器为库的文本和数据段分配空间并映射它们，以及用于 bss 的零填充页面。从库的动态段中，它会将库的符号表添加到符号表链中，如果库需要尚未加载的进一步库，则会将任何新库添加到要加载的列表中。

当此过程终止时，所有库都已映射，并且加载器具有一个逻辑全局符号表，该表由程序和已映射库的所有符号表的并集组成。

共享库初始化

现在加载器会重新访问每个库并处理库的重定位条目，填充库的 GOT 并执行库数据段中所需的任何重定位。x86 上的加载时重定位包括：

- R_386_GLOB_DAT：用于将 GOT 条目初始化为在另一个库中定义的符号的地址。
- R_386_32：对在另一个库中定义的符号的非 GOT 引用，通常是静态数据中的指针。
- R_386_RELATIVE：用于可重定位的数据引用，通常是指向字符串或其他本地定义的静态数据的指针。
- R_386_JMP_SLOT：用于初始化 PLT 的 GOT 条目，稍后描述。

如果库有 .init 节，加载器会调用它进行库特定的初始化，例如 C++ 静态构造函数，并且任何 .fini 节都会被标记为在退出时运行。（它不会对主程序进行初始化，因为这在程序自己的启动代码中处理。）当此阶段完成时，所有库都已完全加载并准备好执行，加载器会调用程序的入口点来启动程序。

指导教师（签字）：_____

年　月　日