7.链接

7.1 编译器驱动程序

大多数编译系统提供编译器驱动程序，它代表用户在需要时调用语言预处理器、编译器、汇编器和链接器。
使用静态链接的方法。

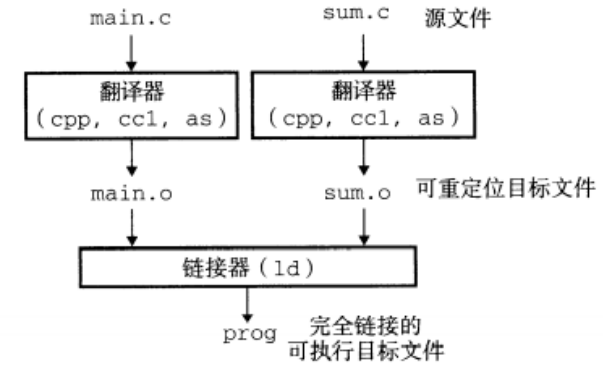


图 7-2　静态链接。链接器将可重定位目标文件组合起来，形成一个可执行目标文件 prog

7.2 静态链接

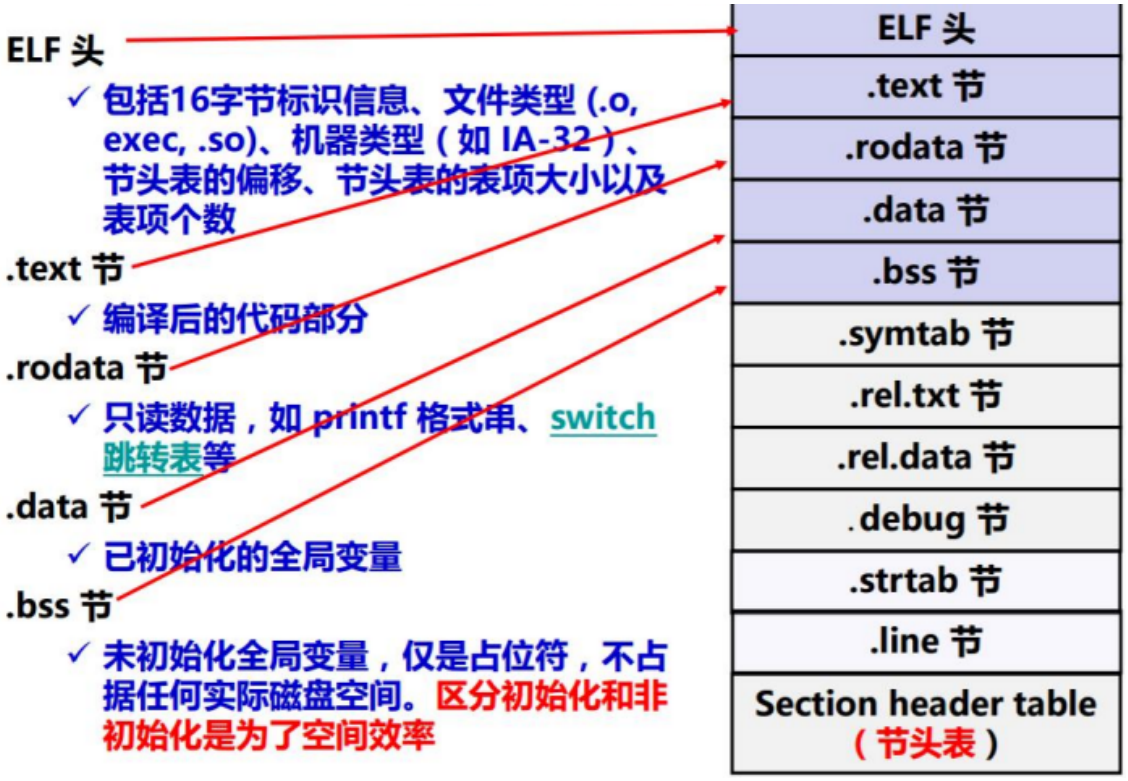
以一组可重定位目标文件和命令行参数作为输入，生成一个完全连接的，可以加载和运行的可执行目标文件作为输出。输入的可重定位目标文件由各种不同的代码和数据节组成。
为了构造可执行文件，连接器必须完成两个主要任务：
1.符号解析。将每个符号引用和一个符号定义关联起来。
2.重定位。编译器和汇编器生成从地址0开始的代码和数据节。连接器通过把每个符号定义与一个内存位置关联起来从而重定位这些节，然后修改所有对这些符号的引用，使得他们指向这个内存位置。

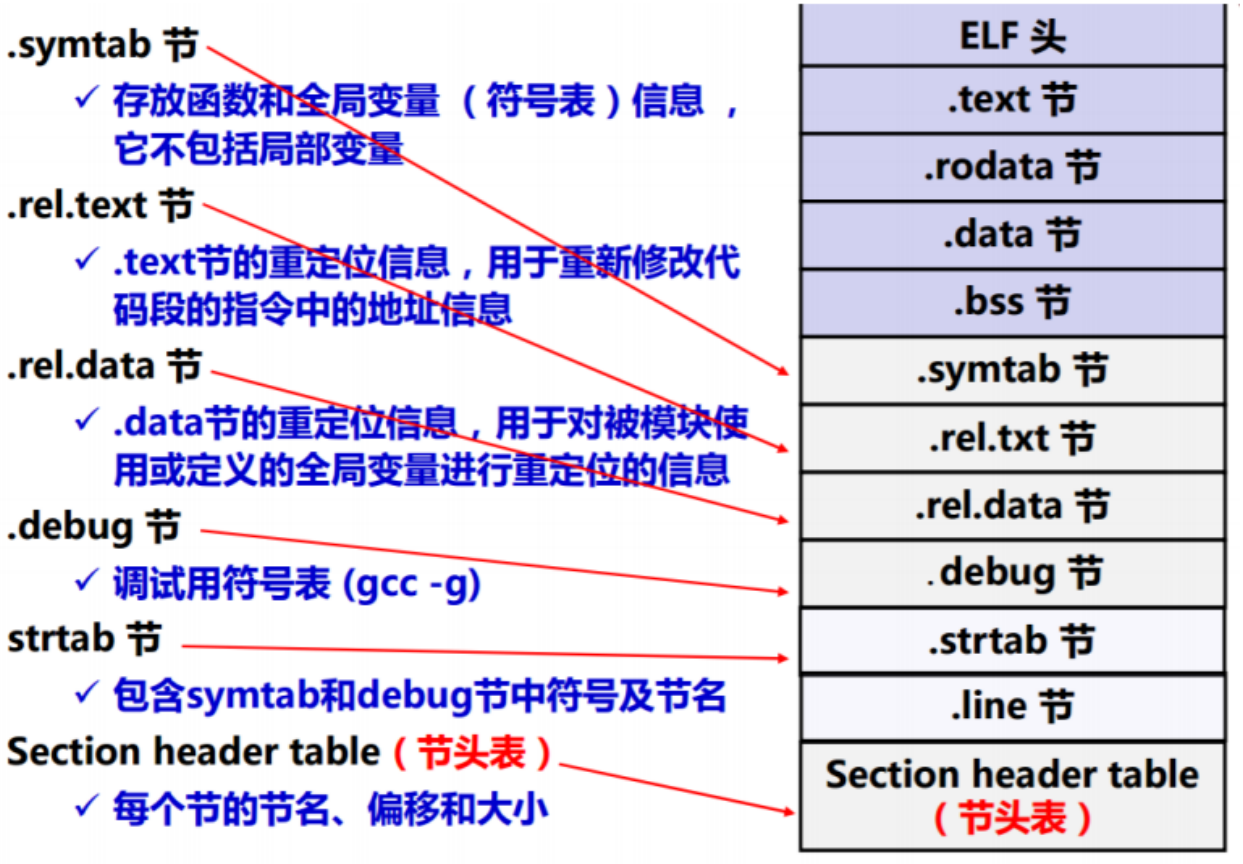
7.3 目标文件

目标文件有三种形式：
1.可重定位目标文件（.o)：包含二进制代码和数据，可以和其他可重定位目标文件合并起来，创建一个可执行目标文件。
2.可执行目标文件（a.out)：包含二进制代码和数据，其形式可以被直接复制到内存并执行。
3.共享目标文件(.so)：一种特殊类型的可重定位目标文件，可以在加载或者运行时被动态地加载进内存并链接。

7.4 可重定位目标文件

现代x86-64 Linux和Unix使用可执行可连接格式。





.symtab**不包含局部变量的条目**

7.5 符号和符号表

首先考虑C中的变量分类：
静态（static）变量相对于给该对象上了一把锁（private），仅限于本文件可见。

No.	分类	作用域	生命期
1	局部变量	所在代码块内	所在函数结束
2	全局变量	所有文件内	程序执行结束
3	静态局部变量	所在代码块内	程序执行结束
4	静态全局变量	当前文件内	程序执行结束
5	普通函数	所有文件内	-
6	静态函数	当前文件内	-

我们将链接器符号分为三类：

- 全局符号
  由模块m定义的，可以被其他模块引用的符号：**普通函数、全局变量**
- 外部符号
  由模块m引用的全局符号，但由其他模块定义
- 局部符号
  由模块m定义和仅由m唯一引用的符号：**静态局部变量、静态全局变量、静态函数**

注意：本地程序的局部变量不属于本地符号！！！因为这类符号是存放在系统运行栈中的，而不是可重定位目标文件中。

符号表是由汇编器构造的，使用编译器输出到汇编语言.s文件中的符号。.symtab节中包含ELF符号表。这张符号表包含一个条目的数组。图7-4展示了每个条目的格式。

code/link/elfstructs.c

```
1  typedef struct {
2      int   name;      /* String table offset */
3      char  type:4,    /* Function or data (4 bits) */
4            binding:4; /* Local or global (4 bits) */
5      char  reserved;  /* Unused */
6      short section;   /* Section header index */
7      long  value;     /* Section offset or absolute address */
8      long  size;      /* Object size in bytes */
9  } Elf64_Symbol;
```

code/link/elfstructs.c

图7-4 ELF符号表条目。type和binding字段每个都是4位

name字段是符号名字在.strtab中的偏移
bingding区分符号是本地或是全局
size是目标的大小
section就是所在节在节头部中的下标
value是相对所在节的起始位置的偏移。对于可执行目标文件来说是一个绝对地址。

三个特殊的伪节在节头部表中是没有条目的。

- ABS表示不该被重定位的符号
- UNDEF表示未定义的符号
- COMMON表示还未被分配位置的未初始化的数据条目。COMMON字段中value表示对齐要求，size给出最小的大小。
  只有可重定位目标文件中才有这些伪节，可执行目标文件中是没有的
  COMMON**中分配未初始化的全局变量，**.bss**中分配未初始化的静态变量以及初始化为0的全局或静态变量**

7.6 符号解析

定义和引用在相同模块中的局部符号的引用，编译器只允许每个模块中每个局部符号有一个定义。静态局部变量也会有本

地连接器符号，编译器要确保他们拥有唯一的名字。

7.6.1 编译器像汇编器输出每个全局符号，或是强或是弱（函数和已初始化的全局变量是强符号，未初始化的全局变量是弱符号），汇编器把这个信息隐含地编码在可重定位目标文件的符号表里。

链接器使用以下规则来处理多重定义的全局符号名：

- 不允许有多个同名的强符号
- 如果一个强符号和多个弱符号同名，选择强符号
- 如果有多个弱符号同名，从这些弱符号中任意选取一个

7.6.2 与静态库链接

将所有相关地目标模块打包成一个单独的文件，称为静态库。

如果不使用库：

- 让编译器辨认出对标准函数的调用并直接生成相应的代码
- 将所有的标准C函数都放在一个单独的可重定位目标模块中。缺点是每个可执行文件中都包含着一份标准函数的副本。
- 为每个标准函数创建一个独立的可重定位文件，把他们存放在一个目录中。但是要求程序员显式地链接合适的目标模块。

静态库解决上述三种方法的缺点。相关的函数可以被编译为独立的目标模块，封装成一个单独的静态库文件。
静态库以一种称为存档的特殊文件格式存放在磁盘中。存档文件是一组连接起来的可重定位目标文件的集合。文件名由后缀.a标识。
链接时只取库中包含被调用的函数的可重定位目标文件。

7.6.3 连接器如何使用静态库来解析引用

在符号解析阶段，链接器从左到右按照他们在编译器驱动程序命令行上出现的顺序来扫锚可重定位目标文件和存档文件。在这次扫描中，链接器维护一个可重定位目标文件的集合E（这个集合中的文件会被合并起来形成可执行文件），一个未解析的符号（即引用了但尚未定义的符号）集合U，以及一个在前面输入文件中已定义的符号D。初始时E、U、D均空。

- 对于命令行中的每个输入文件f，连接器判断f为目标文件还是存档文件。如果是目标文件，将f加入到E中，修改U和D来反映f中的符号定义和引用，并继续下一个输入文件
- 如果f是一个存档文件，连接器尝试匹配U中未解析的符号和由存档文件成员定义的符号。如果某个存档文件成员m定义了一个符号来解析U中的一个引用，那么就将m加到E中，并且修改U和D来反映m中的符号定义和引用。对存档文件中所有的成员目标文件都依次进行这个过程，直到U和D不再发生变化。此时任何不包含在E中的成员目标文件都简单的丢弃，链接器继续处理下一个输入文件。
- 当连接器完成对命令行上输入文件的扫描后，U是非空的，那么连接器就会输出一个错误并终止。否则，他会合并和重定位E中的目标文件，构建输出的可执行文件。
  所以输入文件的顺序会决定结果成功与否。

7.7 重定位

链接器完成了符号解析后就将代码中每个符号引用和一个符号定义关联起来。就可以开始重定位步骤了，这个步骤中，将合并输入模块，并为每个符号分配运行时地址。

重定位由两部分组成：

- 重定位节和符号定义：链接器将所有相同类型的节合并为同一类型的新的聚合节。例如将所有的输入模块的.data节合并成一个节，这个节成为输出的可执行目标文件的.data节。然后，链接器将运行时内存地址赋给新的聚合节，赋给输入模块定义的每个节，以及赋给输入模块定义的每个符号。这步完成后，程序中的每条指令和全局变量都有唯一的运行时内存地址了。
- 重定位界中的符号引用：修改代码节和数据节中对每个符号的引用，使得他们指向正确的运行时地址。依赖于可重定位目标模块中的重定位条目。

7.7.1 重定位条目

汇编器生成一个目标模块时，它并不知道数据和代码最终放在内存中的什么位置，也不知道被他引用的外部定义的函数或全局变量的位置。所以，当汇编器遇到对最终位置未知的目标引用，他就会生成一个重定位条目，告诉链接器将在目标文件合并成可执行文件时如何修改这个引用。代码的重定位条目放在.rel.text中，已初始化数据的重定位条目放在.rel.data中。

code/link/elfstructs.c

```
1  typedef struct {
2      long offset;    /* Offset of the reference to relocate */
3      long type:32,   /* Relocation type */
4           symbol:32; /* Symbol table index */
5      long addend;    /* Constant part of relocation expression */
6  } Elf64_Rela;
```

code/link/elfstructs.c

图 7-9　ELF 重定位条目。每个条目表示一个必须被重定位的引用，并指明如何计算被修改的引用

offset是需要被修改的引用的节内偏移；
symbol表示被修改引用为应指向的符号；
addend是一个有符号常数。
ELF定义了32种重定位类型。我们只关心两种最基本的：

- R_X86_64_PC32：重定位使用32位PC相对地址的引用。
- R_X86_64_32：重定位一个使用32位绝对地址的引用。

7.7.2 重定位符号引用

例子：

PC相对寻址

code/link/main-relo.d

```
1  0000000000000000 <main>:
2     0:   48 83 ec 08             sub    $0x8,%rsp
3     4:   be 02 00 00 00          mov    $0x2,%esi
4     9:   bf 00 00 00 00          mov    $0x0,%edi        %edi = &array
5                          a: R_X86_64_32 array           Relocation entry

6     e:   e8 00 00 00 00          callq  13 <main+0x13>   sum()
7                          f: R_X86_64_PC32 sum-0x4       Relocation entry
8    13:   48 83 c4 08             add    $0x8,%rsp
9    17:   c3                      retq
```

code/link/main-relo.d

图 7-11　main.o 的代码和重定位条目。原始 C 代码在图 7-1 中

相应的重定位条目 r 由 4 个字段组成：

```
r.offset = 0xf
r.symbol = sum
r.type   = R_X86_64_PC32
r.addend = -4
```

这些字段告诉链接器修改开始于偏移量 0xf 处的 32 位 PC 相对引用，这样在运行时它会指向 sum 例程。现在，假设链接器已经确定

```
ADDR(s) = ADDR(.text) = 0x4004d0
```

和

```
ADDR(r.symbol) = ADDR(sum) = 0x4004e8
```

要跳转的指令的地址

使用图 7-10 中的算法，链接器首先计算出引用的运行时地址(第 7 行)：

```
refaddr = ADDR(s)  + r.offset
        = 0x4004d0 + 0xf
        = 0x4004df
```

要修改的引用的运行时地址

然后，更新该引用，使得它在运行时指向 sum 程序(第 8 行)：

```
*refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr)
        = (unsigned) (0x4004e8       + (-4)     - 0x4004df)
        = (unsigned) (0x5)
```

所以本质就是用跳转地址减去下一条指令地址

-4实际上就是计算出跳转指令的下一条指令的地址，

在得到的可执行目标文件中，call 指令有如下的重定位的形式：

```
4004de:  e8 05 00 00 00          callq  4004e8 <sum>      sum()
```

在运行时，call 指令将存放在地址 0x4004de 处。当 CPU 执行 call 指令时，PC 的值为 0x4004e3，即紧随在 call 指令之后的指令的地址。为了执行这条指令，CPU 执行以下的步骤：

1) 将 PC 压入栈中

2) PC ← PC + 0x5 = 0x4004e3 + 0x5 = 0x4004e8

PC绝对寻址

```
r.offset = 0xa
r.symbol = array
r.type   = R_X86_64_32
r.addend = 0
```

这些字段告诉链接器要修改从偏移量 0xa 开始的绝对引用，这样在运行时它将会指向 array 的第一个字节。现在，假设链接器已经确定

```
ADDR(r.symbol) = ADDR(array) = 0x601018
```

链接器使用图 7-10 中算法的第 13 行修改了引用：

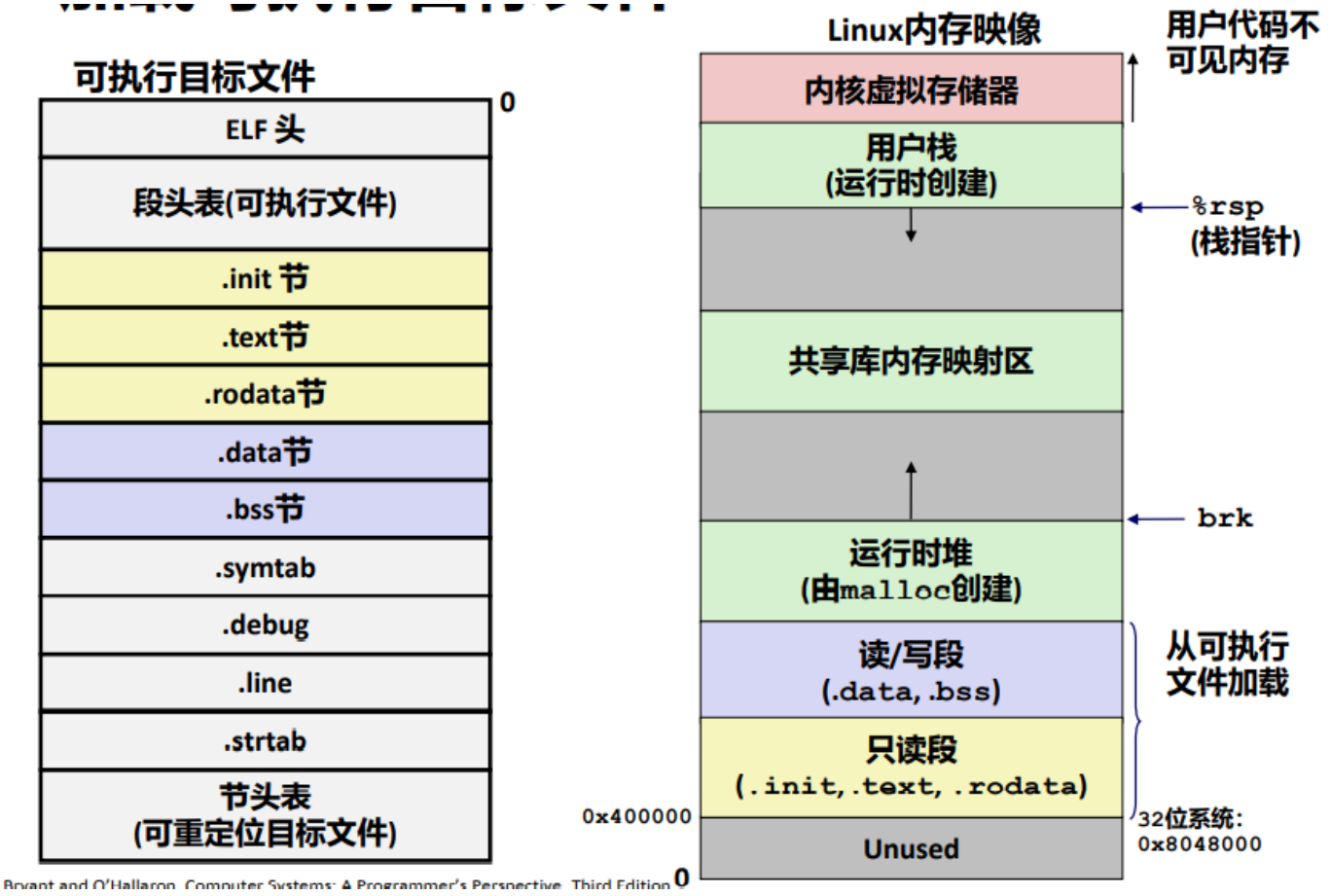
```
*refptr = (unsigned) (ADDR(r.symbol) + r.addend)
        = (unsigned) (0x601018       + 0)
        = (unsigned) (0x601018)
```

在得到的可执行目标文件中，该引用有下面的重定位形式：

```
4004d9:  bf 18 10 60 00          mov    $0x601018,%edi   %edi = &array
```

7.8 可执行目标文件

完成了重定位后，我们就生成了可执行的目标文件，此时可以加载到内存中运行了。



7.9 加载可执行目标文件

```
linux>   ./prog
```

因为prog不是一个内置的shell指令，shell会认为prog是一个可执行目标文件，通过调用某个驻留在存储器中称为加载器的OS代码来运行它。
加载器将可执行目标文件中的代码和数据从磁盘复制到内存中，然后通过跳转到程序的第一条指令或者入口点来运行该程序。

7.10 动态链接共享库

解决静态库的问题，静态库更新后需要重新链接，被重复使用的库函数还是会被重复复置，浪费了内存。
共享库是一个目标模块，在运行或加载时可以加载到任意的内存地址，并和一个在内存中的程序链接起来。称为动态链接。

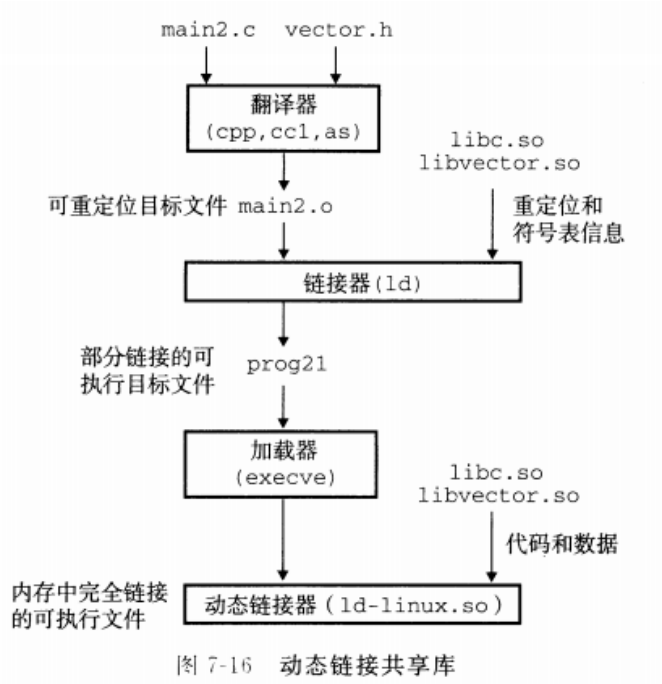


图 7-16 动态链接共享库

没有任何.so的代码和数据节真的被复制到可执行文件中，链接器复制了一些重定位和符号表信息，它们使得运行时可以解析对.so中代码和数据的引用。

- 重定位 libc.so 的文本和数据到某个内存段。
- 重定位 libvector.so 的文本和数据到另一个内存段。
- 重定位 prog21 中所有对由 libc.so 和 libvector.so 定义的符号的引用。

最后，动态链接器将控制传递给应用程序。从这个时刻开始，共享库的位置就固定了，并且在程序执行的过程中都不会改变。

7.11 从应用程序中加载和链接共享库

Linux系统为动态链接器提供了一个简单的接口，允许应用程序在运行时加载和连接共享库。

```
#include <dlfcn.h>

void *dlopen(const char *filename, int flag);
                                        返回：若成功则为指向句柄的指针，若出错则为 NULL。
```

7.12位置无关代码

可以加载而无需重定位的代码称为位置无关代码PIC。使用-fpic选项指示GUN编译系统生成PIC代码。共享库的编译必须总是使用该选项。

1.PIC数据引用

无论我们在内存中的何处加载一个目标模块（包括共享目标模块），数据段与代码段的距离总是保持不变。因此代码段中的任何指令和数据段中任何变量之间的距离是一个运行时常量，与代码段和数据段的绝对内存位置是无关的。

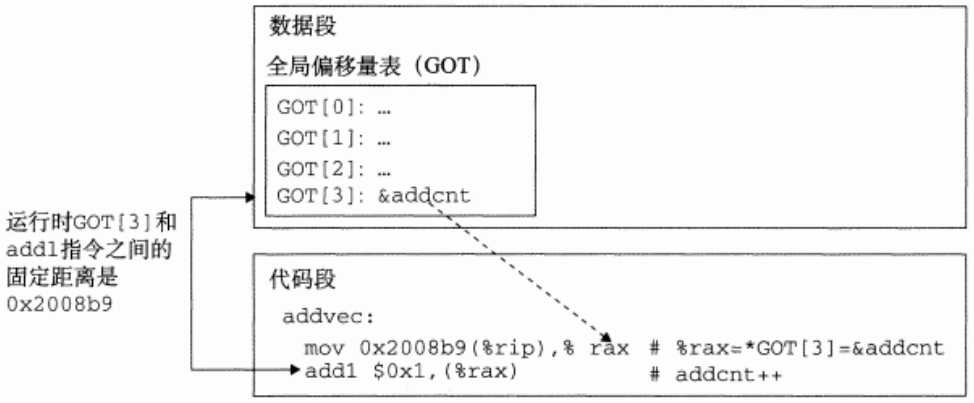
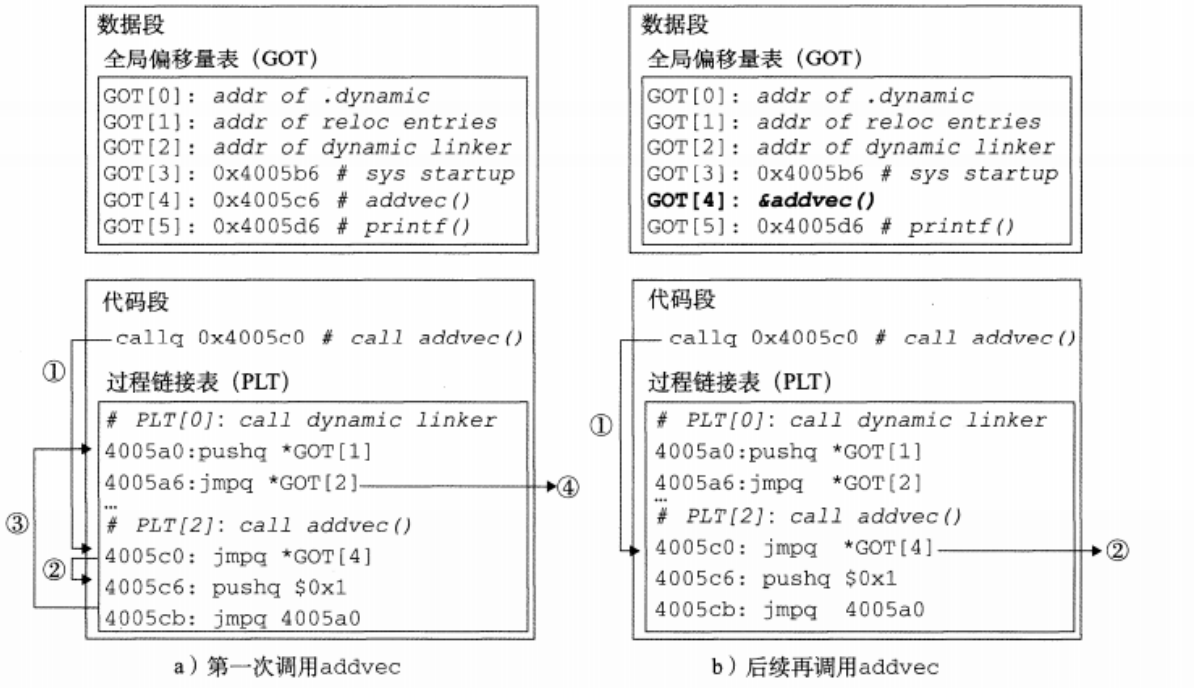


图 7-18 用 GOT 引用全局变量。libvector.so 中的 addvec 例程通过 libvector.so 的 GOT 间接引用了 addcnt

2.PIC函数调用

延迟绑定，将过程地址的绑定推迟到第一次调用该过程时。

每个GOT条目初始时都指向对应PLT条目的第二条指令。



a）第一次调用addvec　　b）后续再调用addvec

图 7-19 用 PLT 和 GOT 调用外部函数。在第一次调用 addvec 时，动态链接器解析它的地址

图 7-19a 展示了 GOT 和 PLT 如何协同工作，在 addvec 被第一次调用时，延迟解析它的运行时地址：

- 第 1 步。不直接调用 addvec，程序调用进入 PLT[2]，这是 addvec 的 PLT 条目。
- 第 2 步。第一条 PLT 指令通过 GOT[4]进行间接跳转。因为每个 GOT 条目初始时都指向它对应的 PLT 条目的第二条指令，这个间接跳转只是简单地把控制传送回 PLT[2]中的下一条指令。
- 第 3 步。在把 addvec 的 ID(0x1)压入栈中之后，PLT[2]跳转到 PLT[0]。
- 第 4 步。PLT[0]通过 GOT[1]间接地把动态链接器的一个参数压入栈中，然后通过 GOT[2]间接跳转进动态链接器中。动态链接器使用两个栈条目来确定 addvec 的运行时位置，用这个地址重写 GOT[4]，再把控制传递给 addvec。

图 7-19b 给出的是后续再调用 addvec 时的控制流：

- 第 1 步。和前面一样，控制传递到 PLT[2]。
- 第 2 步。不过这次通过 GOT[4]的间接跳转会将控制直接转移到 addvec。

7.13 库打桩机制

允许截获对共享库函数的调用，取而代之执行自己的代码。

- 编译时打桩，告诉编译器先使用自己编写的目标文件。
- 链接时打桩

Linux 静态链接器支持用--wrap f 标志进行链接时打桩。这个标志告诉链接器，把对符号 f 的引用解析成__wrap_f(前缀是两个下划线)，还要把对符号__real_f(前缀是两个下划线)的引用解析为 f。图 7-21 给出我们示例程序的包装函数。

```
linux> gcc -Wl,--wrap,malloc -Wl,--wrap,free -o intl int.o mymalloc.o
```

-Wl,option 标志把 option 传递给链接器。option 中的每个逗号都要替换为一个空格。所以-Wl,--wrap,malloc 就把--wrap malloc 传递给链接器，以类似的方式传递-Wl,--wrap,free。

code/link/interpose/mymalloc.c

```
1  #ifdef LINKTIME
2  #include <stdio.h>
3
4  void *__real_malloc(size_t size);
5  void __real_free(void *ptr);
6
7  /* malloc wrapper function */
8  void *__wrap_malloc(size_t size)
9  {
10     void *ptr = __real_malloc(size); /* Call libc malloc */
11     printf("malloc(%d) = %p\n", (int)size, ptr);
12     return ptr;
13 }
14
15 /* free wrapper function */
16 void __wrap_free(void *ptr)
17 {
18     __real_free(ptr); /* Call libc free */
19     printf("free(%p)\n", ptr);
20 }
21 #endif
```

code/link/interpose/mymalloc.c

图 7-21 用--wrap 标志进行链接时打桩

- 运行时打桩。 编译时打桩需要访问程序的源代码，链接时打桩需要访问程序的可重定位对象文件。运行时打桩只要能够访问可执行目标文件。