北京大学 PEKING UNIVERSITY | 物理学院

链接（静态库、重定位、可执行目标文件）

计算机系统导论　课程回课

包涛尼

北京大学物理学院

2022 年 11 月 9 日

Contents

1. 静态库
2. 重定位
3. 可执行目标文件

静态库

为什么需要静态库?

- 迄今为止，我们都是假设链接器读取一组可重定位目标文件，并把它们链接起来，形成一个输出的可执行文件.
- 但所有的编译系统都提供一种机制，将所有相关的目标模块打包成为一个单独的文件，称为静态库（static library），它可以用做链接器的输入. 当链接器构造一个输出的可执行文件时，它只复制静态库里被应用程序引用的目标模块.
- 相关的函数可以被编译为独立的目标模块，然后封装成一个单独的静态库文件. 然后，应用程序可以通过在命令行上指定单独的文件名字来使用这些在库中定义的函数. 比如，使用 C 标准库和数学库中函数的程序可以用形式如下的命令行来编译和链接：

```
linux> gcc main.c /usr/lib/libm.a /usr/lib/libc.a
```

- 在链接时，链接器将只复制被程序引用的目标模块，这就减少了可执行文件在磁盘和内存中的大小. 另一方面，应用程序员只需要包含较少的库文件的名字.

与静态库链接

- 在 Linux 系统中，静态库以一种称为存档（archive）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，有一个头部用来描述每个成员目标文件的大小和位置. 存档文件名由后级 .a 标识.

code/link/addvec.c

```
1  int addcnt = 0;
2
3  void addvec(int *x, int *y,
4              int *z, int n)
5  {
6      int i;
7
8      addcnt++;
9
10     for (i = 0; i < n; i++)
11         z[i] = x[i] + y[i];
12 }
```

code/link/addvec.c

a) addvec.o

code/link/multvec.c

```
1  int multcnt = 0;
2
3  void multvec(int *x, int *y,
4               int *z, int n)
5  {
6      int i;
7
8      multcnt++;
9
10     for (i = 0; i < n; i++)
11         z[i] = x[i] * y[i];
12 }
```

code/link/multvec.c

b) multvec.o

图 1 libvector 库中的成员目标文件

与静态库链接

- 创建这些函数的一个静态库（使用 AR 工具）:

```
linux> gcc -c addvec.c multvec.c
linux> ar rcs libvector.a addvec.o multvec.o
```

code/link/main2.c

```
1   #include <stdio.h>
2   #include "vector.h"
3
4   int x[2] = {1, 2};
5   int y[2] = {3, 4};
6   int z[2];
7
8   int main()
9   {
10      addvec(x, y, z, 2);
11      printf("z = [%d %d]\n", z[0], z[1]);
12      return 0;
13  }
```

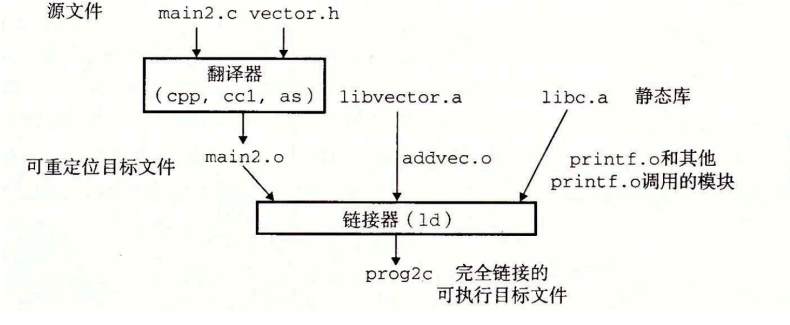
code/link/main2.c

与静态库链接

- 为了创建这个可执行文件，我们要编译和链接输入文件 `main.o` 和 `libvector.a`：

```
linux> gcc -c main2.c
linux> gcc -static -o prog2c main2.o ./libvector.a
```

- `-static` 参数告诉编译器驱动程序，链接器应该构建一个完全链接的可执行目标文件，它可以加载到内存并运行，在加载时无须更进一步的链接.



链接器使用静态库解析引用

- 在符号解析阶段，链接器从左到右按照它们在编译器驱动程序命令行上出现的顺序来扫描可重定位目标文件和存档文件.（驱动程序自动将命令行中所有的 `.c` 文件翻译为 `.o` 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（即引用了但是尚未定义的符号）集合 U，以及一个在前面输入文件中已定义的符号集合 D. 初始时，E、U 和 D 均为空.

链接器使用静态库解析引用

1. 对于命令行上的每个输入文件 f，链接器会判断 f 是一个目标文件还是一个存档文件. 如果 f 是一个目标文件，那么链接器把 f 添加到 E，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.
2. 如果 f 是一个存档文件，那么链接器就尝试匹配 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，并且链接器修改 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 都不再发生变化. 此时，任何不包含在 E 中的成员目标文件都简单地被丢弃，而链接器将继续处理下一个输入文件.
3. 如果当链接器完成对命令行上输入文件的扫描后，U 是非空的，那么链接器就会输出一个错误并终止. 否则，它会合并和重定位 E 中的目标文件，构建输出的可执行文件.

链接器使用静态库解析引用

- 会发生什么?

```
linux> gcc -static ./libvector.a main2.c
```

- 注意：在命令行中，如果定义一个符号的库出现在引用这个符号的目标文件之前，那么引用就不能被解析，链接会失败.
- 关于库的一般准则是将它们放在命令行的结尾. 如果各个库的成员是相互独立的（也就是说没有成员引用另一个成员定义的符号），那么这些库就可以以任何顺序放置在命令行的结尾处. 另一方面，如果库不是相互独立的，那么必须对它们排序，使得对于每个被存档文件的成员外部引用的符号 s，在命令行中至少有一个 s 的定义是在对 s 的引用之后的.

重定位

重定位条目

- 重定位由两步组成：
  1. **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.
  2. **重定位节中的符号引用**：链接器修改代码节和数据节中对每个符号的引用，使得它们指向正确的运行时地址.
- ELF 重定位条目：`offset`：需要被修改的引用的节偏移；`symbol`：被修改引用应该指向的符号；`type`：链接器如何修改新的引用；`addend`：有符号常数，一些类型的重定位要使用它对被修改引用的值做偏移调整.

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

重定位符号引用

- 假设每个节 `s` 是一个字节数组，每个重定位条目 `r` 是一个类型为 `Elf64_Rela` 的结构．运行时地址用 `ADDR()` 表示．重定位算法可以被描述如下：

```
1  foreach section s {
2      foreach relocation entry r {
3          refptr = s + r.offset;
4          if (r.type == R_X86_64_PC32) { // PC 相对寻址
5              refaddr = ADDR(s) + r.offset;
6              *refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr);
7          }
8          if (r.type == R_X86_64_32) // 绝对寻址
9              *refptr = (unsigned) (ADDR(r.symbol) + r.addend);
10     }
11 }
```

重定位符号引用

- 考虑如下示例：

code/link/main.c

```
1  int sum(int *a, int n);
2
3  int array[2] = {1, 2};
4
5  int main()
6  {
7      int val = sum(array, 2);
8      return val;
9  }
```

code/link/main.c

a) main.c

code/link/sum.c

```
1  int sum(int *a, int n)
2  {
3      int i, s = 0;
4
5      for (i = 0; i < n; i++) {
6          s += a[i];
7      }
8      return s;
9  }
```

code/link/sum.c

b) sum.c

图 2 main 函数初始化一个整数数组，然后调用 sum 函数来对数组元素求和

重定位符号引用

- 考虑如下示例：

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 3 main.o 的代码和重定位条目

重定位 PC 相对引用

- call 指令开始于节偏移 Oxe 的地方，包括 1 字节的操作码 0×e8，后面跟着的是对目标 sum 的 32 位 PC 相对引用的占位符.
- 相应的重定位条目 r 由 4 个字段组成：

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

- 假设链接器已经确定

```
ADDR(s) = ADDR(.text)= 0x4004d0
ADDR(r.symbol) = ADDR(sum) = 0x4004e8
```

重定位 PC 相对引用

- 链接器首先计算出引用的运行时地址：

```
refaddr = ADDR(s) + r.offset
        = 0x4004a0 + 0xf
        = 0x4004df
```

- 更新该引用，使得它在运行时指向 sum 程序

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

- call 有如下重定位格式：

```
4004de:     85 05 00 00 00      callq   4004e8 <sum>
```

重定位绝对引用

- mov 指令开始于节偏移 0x9 的地方，包括 1 字节的操作码 0×bf，后面跟着的是对 array 的 32 位绝对引用的占位符.
- 相应的重定位条目 r 由 4 个字段组成：

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

- 假设链接器已经确定

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

重定位绝对引用

- 更新该引用

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

- 该饮用有如下重定位格式：

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

重定位

（2019 年期末）在链接时，对于下列哪些符号需要进行重定位?

(1) 不同 C 语言源文件中定义的函数

(2) 同一 C 语言源文件中定义的全局变量

(3) 同一函数中定义时不带 `static` 的变量

(4) 同一函数中定义时带有 `static` 的变量

A. (1)(3)　　B. (2)(4)　　C. (1)(2)(4)　　D. (1)(2)(3)(4)

重定位

（2019 年期末）在链接时，对于下列哪些符号需要进行重定位?

(1) 不同 C 语言源文件中定义的函数

(2) 同一 C 语言源文件中定义的全局变量

(3) 同一函数中定义时不带 `static` 的变量

(4) 同一函数中定义时带有 `static` 的变量

A. (1)(3)　　B. (2)(4)　　C. (1)(2)(4)　　D. (1)(2)(3)(4)

C. 例如全局变量如果初始化为值且不使用，则不需要重定位. 需要留意，如果是同一 C 语言源文件中定义的函数则一般不需要重定位.

可执行目标文件

可执行目标文件

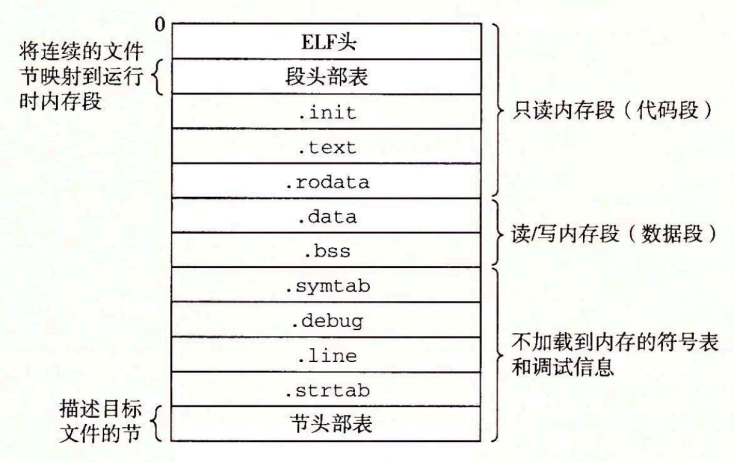


图 4 典型的 ELF 可执行目标文件

加载可执行目标文件

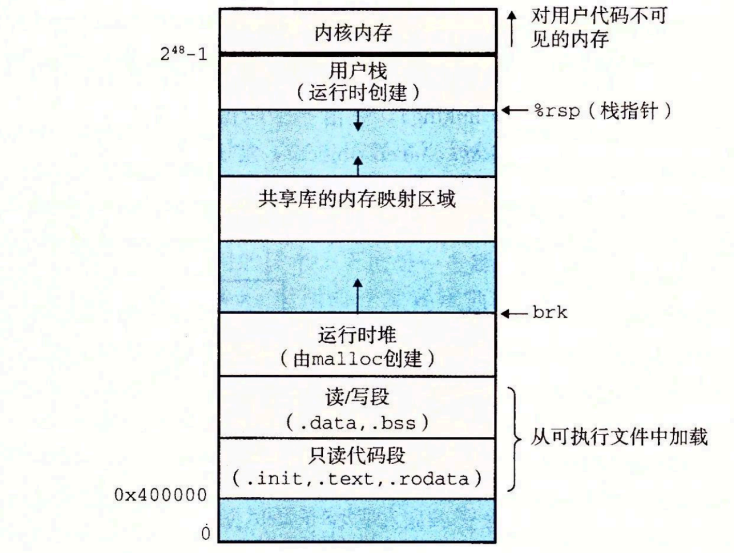


图 5　Linux x86-64 运行时内存映像

参考文献

[1] Bryant R E, O'Hallaron D R. 深入理解计算机系统 [M]. 北京: 机械工业出版社, 2016.

谢 谢！