静态链接器是由像 GCC 这样的编译驱动程序调用的。它们将多个可重定位目标文件合并成一个单独的可执行目标文件。多个目标文件可以定义相同的符号，而链接器用来悄悄地解析这些多重定义的规则可能在用户程序中引入微妙的错误。

多个目标文件可以被连接到一个单独的静态库中。链接器用库来解析其他目标模块中的符号引用。许多链接器通过从左到右的顺序扫描来解析符号引用，这是另一个引起令人迷惑的链接时错误的来源。

加载器将可执行文件的内容映射到内存，并运行这个程序。链接器还可能生成部分链接的可执行目标文件，这样的文件中有对定义在共享库中的例程和数据的未解析的引用。在加载时，加载器将部分链接的可执行文件映射到内存，然后调用动态链接器，它通过加载共享库和重定位程序中的引用来完成链接任务。

被编译为位置无关代码的共享库可以加载到任何地方，也可以在运行时被多个进程共享。为了加载、链接和访问共享库的函数和数据，应用程序也可以在运行时使用动态链接器。

参考文献说明

在计算机系统文献中并没有很好地记录链接。因为链接是处在编译器、计算机体系结构和操作系统的交叉点上，它要求理解代码生成、机器语言编程、程序实例化和虚拟内存。它没有恰好落在某个通常的计算机系统领域中，因此这些领域的经典文献并没有很好地描述它。然而，Levine 的专著提供了有关这个主题的很好的一般性参考资料[69]。[54]描述了 ELF 和 DWARF(对 .debug 和 .line 节内容的规范)的原始 IA32 规范。[36]描述了对 ELF 文件格式的 x86-64 扩展。x86-64 应用二进制接口(ABI)描述了编译、链接和运行 x86-64 程序的惯例，其中包括重定位和位置无关代码的规则[77]。

家庭作业

*7.6 这道题是关于图 7-5 的 m.o 模块和下面的 swap.c 函数版本的，该函数计算自己被调用的次数：

```
1   extern int buf[];
2
3   int *bufp0 = &buf[0];
4   static int *bufp1;
5
6   static void incr()
7   {
8       static int count=0;
9
10      count++;
11  }
12
13  void swap()
14  {
15      int temp;
16
17      incr();
18      bufp1 = &buf[1];
19      temp = *bufp0;
20      *bufp0 = *bufp1;
21      *bufp1 = temp;
22  }
```

对于每个 swap.o 中定义和引用的符号，请指出它是否在模块 swap.o 的 .symtab 节中有符号表条目。如果是这样，请指出定义该符号的模块(swap.o 或 m.o)、符号类型(局部、全局或外部)以及它在模块中所处的节(.text、.data 或 .bss)。