Bonus 2：链接使用共享库的程序

在Bonus 1中，你学会了如何生成共享库。现在让我们站在使用者的角度，看看程序如何调用共享库中的函数。这个任务会引入动态链接中最核心的两个机制：全局偏移表（GOT）和过程链接表（PLT）。

运行时地址的不确定性

假设你的程序需要调用共享库中的函数`foo`。在编译时，编译器生成了一条`call`指令，但这条指令的目标地址该填什么呢？

程序不知道`foo`会在运行时被加载到什么地址。这取决于操作系统如何安排内存布局，可能每次运行都不一样。你在CSAPP第九章中应该学过，现代系统使用地址空间随机化（ASLR）来增强安全性，共享库的加载位置是随机的。

一个直观的想法是：程序先正常运行，当加载器加载所有共享库后，找到`foo`的实际地址，然后修改程序中所有调用`foo`的指令，把正确的地址填进去。这种方法可行，但有个问题——它需要修改代码段的内容。

你还记得我们在任务六中设置的权限吗？代码段应该是只读的，运行时修改它违反了内存保护原则。更重要的是，如果代码段可写，攻击者可能利用漏洞修改程序的指令，这是许多安全问题的根源。

引入间接层：全局偏移表

更好的方法是引入一个间接层。我们在数据段中创建一个表，叫做全局偏移表（Global Offset Table, GOT）。这个表中的每个条目是一个指针，指向一个外部符号的实际地址。

当程序需要调用`foo`时，它不是直接跳转到`foo`的地址，而是先查GOT表找到`foo`的地址，然后跳转到那个地址。代码本身不需要修改，所有的地址填充工作都发生在GOT表中——而GOT表位于可读写数据段，加载器可以安全地更新它。

你可以把GOT理解为一个"地址簿"。程序说"我要找foo"，GOT回答"foo在0x7ffff7a12340"。程序按照这个地址跳转，就找到了正确的函数。

位置无关代码的要求

在深入实现细节之前，需要明确一个重要的前提：本任务中，输入的目标文件必须使用位置无关代码编译（`-fPIC`选项）。

当编译器以PIC模式编译代码时，它会为外部符号的引用生成特殊的指令序列和重定位类型。比如：

对于外部函数调用，编译器生成的重定位类型是`R_X86_64_PLT32`，而不是普通的`R_X86_64_PC32`。这告诉链接器："这个调用应该通过PLT进行"。

对于外部数据访问，编译器会生成两步访问的代码序列，并使用`R_X86_64_GOTPCREL`重定位。比如访问`extern int var`时：

```
mov rax, [rip + offset]   ; 从GOT读取var的地址（带GOTPCREL重定位）
mov eax, [rax]            ; 从该地址读取var的值
```

第一条指令的重定位由链接器处理，让offset指向GOT中`var`对应的槽位。第二条指令在运行时从GOT读到的地址中取值。

这个设计很巧妙：编译器负责生成正确的指令序列（知道需要两步访问），链接器负责分配GOT空间并正确处理重定位，加载器负责填充GOT。每个组件做好自己的工作，不需要链接器去改写编译器生成的指令。

> **Note**
> 在本任务中，我们主要关注外部函数调用的处理。外部数据的访问由编译器的PIC代码生成机制处理——编译器会生成正确的指令序列和`GOTPCREL`重定位，链接器只需要正确处理这种重定位类型，让它指向GOT即可。

函数调用的特殊挑战

现在让我们专注于函数调用。虽然编译器已经生成了`R_X86_64_PLT32`重定位，但这里仍然有一个技术问题需要解决。

编译器生成的是直接调用指令：

```
e8 [4-byte offset]  ; call 指令
```

这条指令的语义是"跳转到当前IP + offset的位置并执行那里的代码"。它期望跳转到一段可执行代码，而不是数据。

如果我们让这条指令直接跳到GOT表的某个条目，会发生什么？GOT表里存的是一个地址值（比如`0x7ffff7a12340`），是数据而不是指令。CPU会尝试把这个64位地址当作机器指令来解码执行，结果会是非法指令错误或段错误，程序崩溃。

这个问题的根源在于，`call`指令是直接调用，它要求目标是可执行代码。但我们需要的是间接调用——先读取一个地址，再跳转到那个地址。x86-64有间接调用指令（比如`call *(%rax)`），但这需要编译器生成不同的指令序列，而且会占用寄存器。

过程链接表：从直接到间接的桥梁

解决方案是过程链接表（Procedure Linkage Table, PLT）。对于每个外部函数，链接器在代码段中生成一小段"桩代码"（stub）。这段代码非常简单，只做一件事：从GOT读取地址，然后跳转到那个地址。

对于x86-64架构，这段桩代码的机器码是：

```
ff 25 [4-byte offset]  ; jmp *offset(%rip)
```

这是一个间接跳转指令。它从指定的内存位置（用RIP相对寻址）读取一个地址，然后跳转到那个地址。我们让这个内存位置指向GOT表中`foo`对应的条目。

具体来说，这6个字节的含义是：

- `ff 25`：这是x86-64的间接跳转指令编码。第一个字节`ff`是操作码前缀，第二个字节`25`指定了具体的跳转形式（RIP相对的间接跳转）。
- 接下来的4个字节是有符号偏移量，小端序。这个偏移是从下一条指令的位置（也就是stub结束位置）到GOT条目的距离。

这里有一个需要注意的细节：x86-64的RIP相对寻址中，偏移量是相对于当前指令之后的位置，而不是当前指令的开始。所以如果你的PLT stub从地址`stub_addr`开始，指令占6字节，那么计算offset的公式是：

```
offset = got_entry_addr - (stub_addr + 6)
```

现在，整个调用链变成了：

1. 程序执行`call foo_stub`（直接调用桩代码，由链接器重定位`PLT32`实现）
2. 桩代码执行`jmp *GOT[foo]`（从GOT读取地址并间接跳转）
3. 跳转到共享库中`foo`的实际位置

这样，原本的直接调用被转换成了通过桩代码的间接调用。桩代码位于程序的代码段，是只读的。GOT表位于数据段，可以被加载器修改。完美地分离了"需要修改的部分"和"不能修改的部分"。

> **Note**
> 在标准的ELF动态链接中，PLT还承担另一个职责——延迟绑定（lazy binding）。第一次调用一个函数时，PLT不是直接跳到GOT，而是跳到一个解析器，解析器查找函数地址，更新GOT，然后再跳转。这样，只有实际被调用的函数才会被解析，加快了程序启动速度。
>
> 在我们的实验中，我们使用更简单的立即绑定（eager binding）——程序启动时就解析所有外部符号，填充GOT表。这样可以避免实现解析器的复杂性，专注于理解GOT/PLT的基本机制。从链接器的角度，两种方案的区别只在于GOT的初始值（延迟绑定需要初始指向解析器），生成PLT stub的过程是相同的。

实现策略

链接使用共享库的程序，需要在静态链接的基础上增加几个步骤。整体流程是：先像静态链接那样处理，然后为外部引用创建GOT和PLT结构。

第一步，像往常一样处理所有输入的目标文件，合并节，构建符号表。这一步和静态链接相同。

第二步，识别外部符号。在处理重定位时，检查每个重定位项的目标符号。如果符号在当前链接的目标文件或静态库中定义，说明是内部符号，正常处理。如果符号未定义，说明是外部符号，需要通过动态链接解析。你可以维护一个外部符号的集合，记录所有需要动态解析的符号。

第三步，为外部符号分配资源。首先为每个外部符号在GOT中分配一个槽位。GOT本质上是一个数据节，每个槽位是8字节（存储一个64位地址）。你可以创建一个名为`.got`的节，大小是外部符号数量乘以8。然后，对于每个外部函数（需要被`call`指令调用的符号），还需要为它生成一个PLT stub。

第四步，生成PLT stub。对于每个外部函数，在代码段（或者专门的`.plt`节）中生成6字节的桩代码。你可以使用框架提供的helper函数：

```
std::vector<uint8_t> stub = generate_plt_stub(got_offset);
```

这个函数接收GOT条目相对于PLT stub的偏移（注意是相对于stub结束位置的偏移），返回6字节的机器码。如果你想挑战自己，也可以尝试手工构造这6个字节——第一第二字节固定为`ff 25`，后面4字节是小端序的有符号偏移量。

第五步，重定向原始的重定位。这是关键的一步。当处理对外部符号的重定位时：

- 如果是`R_X86_64_PLT32`重定位（函数调用），计算从当前位置到对应PLT stub的相对偏移，填入那个位置。这样原本的`call external_func`变成了`call plt_stub`。
- 如果是`R_X86_64_GOTPCREL`重定位（数据访问），计算从当前位置到对应GOT条目的相对偏移，填入那个位置。编译器已经生成了正确的两步访问代码，链接器只需要让第一步指向GOT即可。

第六步，生成动态重定位表。对于GOT中的每个条目，创建一个动态重定位项，告诉加载器"请将符号X的地址填入GOT的第Y个槽位"。这个表会被写入输出文件的元数据中。

最后，记录依赖的共享库列表。可执行文件需要知道它依赖哪些库，加载器才知道要加载什么。在输出文件中添加一个`needed`字段，列出所有需要的库名（比如`["libfoo.so", "libc.so"]`）。

> **Tip**
> 在调试时，详细的日志会很有帮助。你可以打印每个外部符号及其在GOT中的索引、在PLT中的stub地址。验证每个原本指向外部符号的重定位是否正确地重定向到了PLT或GOT。如果某个函数调用在运行时崩溃，很可能是PLT stub的offset计算错误，或者GOT条目没有正确填充。

整体流程回顾

让我们用一个完整的例子串联整个过程：

源代码：

```
extern int foo(int x);  // 在共享库中定义
int main() {
    return foo(42);
}
```

编译后（使用`-fPIC`），目标文件包含：

- 一段代码，其中有`call`指令
- 一个`R_X86_64_PLT32`重定位项，说"这里需要`foo`的PLT入口"

链接器处理：

- 发现`foo`是外部符号，在GOT中分配槽位0（地址假设为`0x403000`）
- 生成PLT stub：`ff 25 [offset]`，假设stub位于地址`0x400500`
    - 计算offset：`0x403000 - (0x400500 + 6) = 0x2AFA`
    - 生成机器码：`ff 25 fa 2a 00 00`（小端序）
- 修正`call`指令的重定位，让它跳到`0x400500`（stub地址）
- 生成动态重定位：`GOT[0] = foo的运行时地址`

加载器执行：

- 加载程序和共享库
- 在共享库中查找符号`foo`，假设地址是`0x7ffff7a12340`
- 将`0x7ffff7a12340`写入`0x403000`（GOT[0]）

程序运行：

- 执行`call 0x400500`（跳到stub）
- Stub执行`jmp *0x403000`（间接跳转）
- 从GOT读到`0x7ffff7a12340`，跳转
- 进入共享库中的`foo`函数

完成后，运行测试来验证：

```
make test_bonus2
```

反思：动态链接的权衡

通过完成这两个bonus任务，你已经理解了动态链接的完整流程。现在可以反思一下这个设计的权衡。

动态链接带来了灵活性。库可以独立更新，程序自动获益——至少在接口兼容的前提下。多个程序可以共享同一份库文件（虽然在我们的简化版本中，内存中仍然有多份代码副本）。程序可以在运行时选择加载哪些库，支持插件式的架构。

但动态链接也有代价。每次函数调用都多了一次间接访问——查GOT表。虽然现代CPU的缓存使这个开销很小，但仍然存在。程序启动时需要加载器解析符号，增加了启动延迟。更重要的是，程序的行为部分取决于运行时环境——如果系统的库版本不对，程序可能无法运行或行为异常。这就是所谓的"依赖地狱"。

这些都是工程上的权衡。理解了链接过程后，你能够更好地评估何时应该使用静态链接（追求性能或独立性），何时应该使用动态链接（追求灵活性或资源节约）。没有绝对的好坏，只有在特定场景下的合适选择。

> **Note**
> 如果你想进一步了解标准ELF格式的动态链接机制，可以注意这些术语的对应关系：我们的动态重定位表中的条目，在ELF中对应R_X86_64_JUMP_SLOT（函数）和R_X86_64_GLOB_DAT（数据）类型的重定位。我们的简化版本直接在重定位项中存储符号名字，而ELF通过符号表索引来引用，这是为了效率。PLT的结构在标准ELF中也更复杂，包含PLT0（公共解析入口）和每个函数的PLT条目。我们的实验采用了最简化的实现，专注于理解核心机制。

挑战：改造Bonus 1为完整的PIC

如果你想进一步挑战自己，可以尝试将Bonus 1改造为完整的位置无关代码版本。核心思路是让共享库也使用GOT和PLT来访问外部符号，而不是在加载时修改代码段。

具体来说，你需要为共享库创建自己的GOT节。当库内代码需要访问外部符号时，通过这个GOT间接访问。如果库内代码调用外部函数，还要为共享库生成PLT stub，库内的`call`指令重定向到这些stub，stub再通过GOT跳转。生成的动态重定位不再修改代码段，而只修改GOT（数据段）。

这样改造后，共享库的代码段在运行时保持完全只读，可以在多个进程之间真正共享物理内存。这就是真实世界中所有共享库采用的方案。从概念上说，这和Bonus 2非常相似——都是创建GOT、生成PLT、重定向引用。区别只在于服务的对象：Bonus 2的GOT/PLT是给主程序用的，改造后的Bonus 1的GOT/PLT是给库自己用的。

如果你实现了这个版本，可以通过Pull Request分享你的工作。这不仅会加深你对动态链接的理解，也能帮助后续的同学学习。