链接 & 加载

OS 备胎组

南京大学计算机软件研究所

Shell 再回首

当我们在 shell 中执行 `a.out` 时候，发生了什么?

- fork + execve
- （生成新进程 + 加载程序）

那么，之后呢?

- 操作系统究竟是如何开始执行程序的?

开始执行程序

程序 = 代码 + 数据；执行程序，那么找到第一条执行地址呗

- 可是第一条指令在哪里？？ main 函数吗？可是 main 函数的地址是什么

Naive 方案：我们可以制定规则

- 大佬说：所有的程序第一条地址都必须在 0x12345678
- 操作系统说：了解！我加载程序，把 `$rip` 设置为 0x12345678，起飞！

不够灵活

- 如果不同操作系统想做点其它事呢?
- 除了起始地址外，我们如果想支持其它功能呢?

ELF 文件

制定一个可扩展的标准吧!

> The ELF standard is intended to streamline software development by providing developers with a set of binary interface definitions that extend across multiple operating environments...

ELF 文件描述了一个可执行文件的信息，代码、数据的布局

- 其中包括第一条指令的地址
- 更详细的内容会在之后说明

寻找第一条指令的地址

我们先考虑简单的情况，程序通过静态链接得到 `-static`:

- gcc -static demo.c -o static-demo

我们用 `readelf` 工具开查看 ELF 文件：

- readelf -h static-demo

  ELF Header:

  ...

  Entry point address: 0x400890

  ...

Bingo！然后动动手，眼见为实

起始地址探索

人民群众的老朋友 --- GDB；一个起始地址插入断点 trick

- `br *0/starti`

第一条指令不是从 main 函数开始的

```
|_start
|  -> __libc_start_main
|    -> generic_start_main
|      -> ...
|      -> main
```

原来在 main 函数之前有很多东西需要准备

- 在 C++ 下情况更复杂，之后有更多探索

如果不是静态链接呢？

通过 gcc demo.c -o default-demo 编译，重复我们之前的步骤：

- readelf 查看 `Entry point address: 0x620`
- 然后再用 GDB 观察

（难受，为什么）

整个程序的入口地址都被动态链接了！

- Position Independent Executable

要理解这些，我们得先从动态链接开始了解

假设你对动态链接一无所知，
我们现在来构想一种实现方式

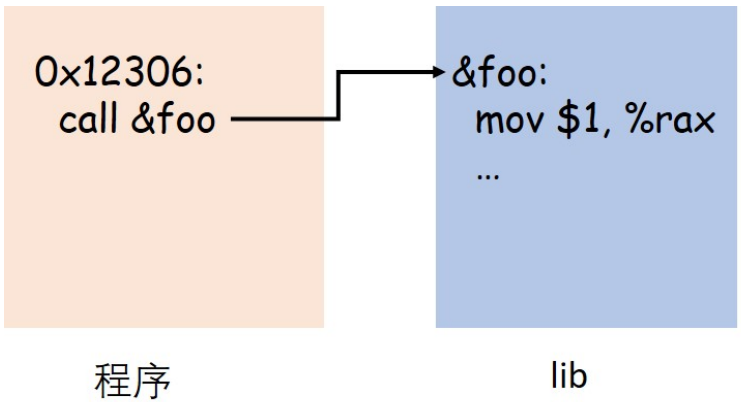
动态链接

为什么需要动态链接?

- 初衷：共享代码

由于代码是动态链接的，所以在无法事先得知代码的的具体地址，

- 如何正确地执行 foo 函数?



动态链接（续）

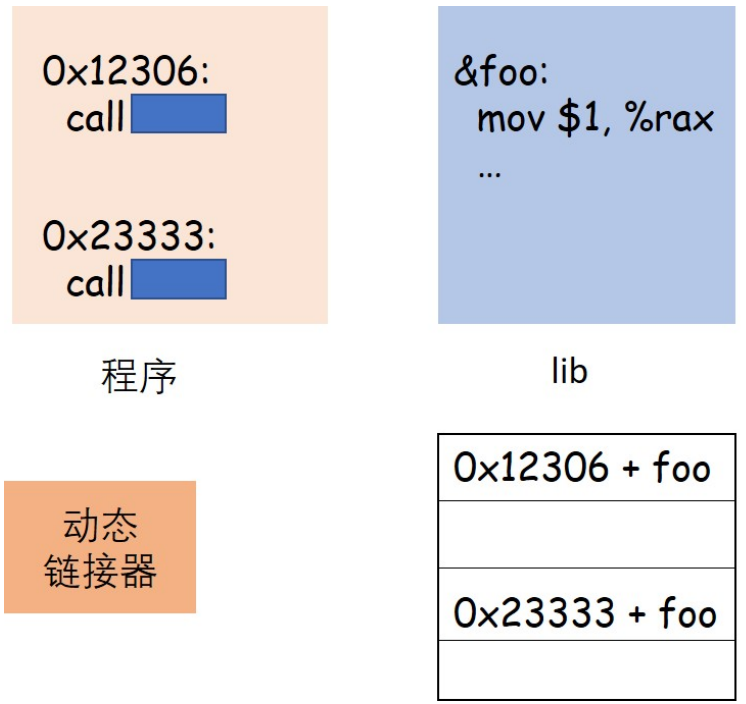
既然事先不能得知地址，那么就留空，让链接器在运行时填空

- 但我们需要知道填空填在哪里，用表格记录（Table）

So far, so good

你可能发现了，这个表格很有很多冗余

- 如果频繁调用同一个函数，表项太多



增加一个间接层

软工名言：

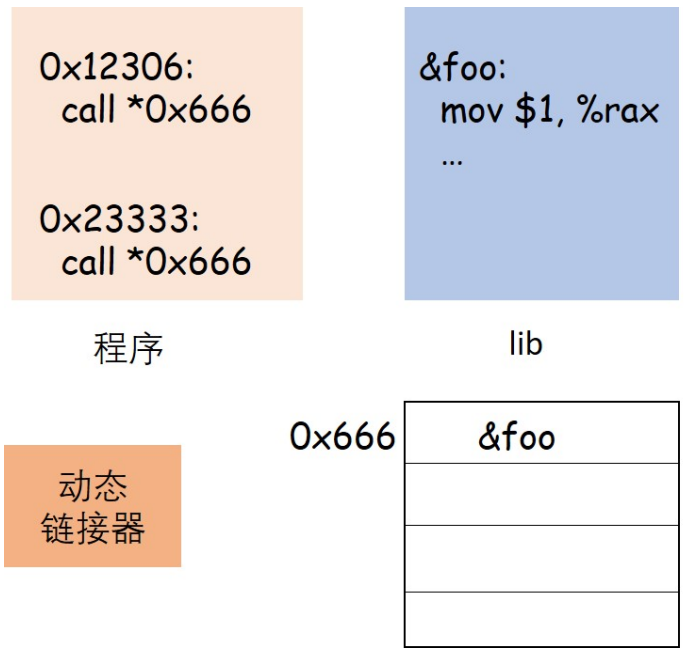
> All problems in computer science can be solved by another level of indirection.
>
> ---David Wheeler

间接出奇迹！

新的记录 Table

我们不在直接地址调用，而是地址间接调用

- 链接器只要修改 &foo 项即可
- `call $addr` -> `call*$addr`



让链接器更完美

问题：虽然一个程序中会调用很多函数，但可能一次执行中只会调用少部分函数

- 只调用执行路径上的函数
- 如果把没执行的函数也进行填表，浪费！

Lazy evaluation（惰性求值）

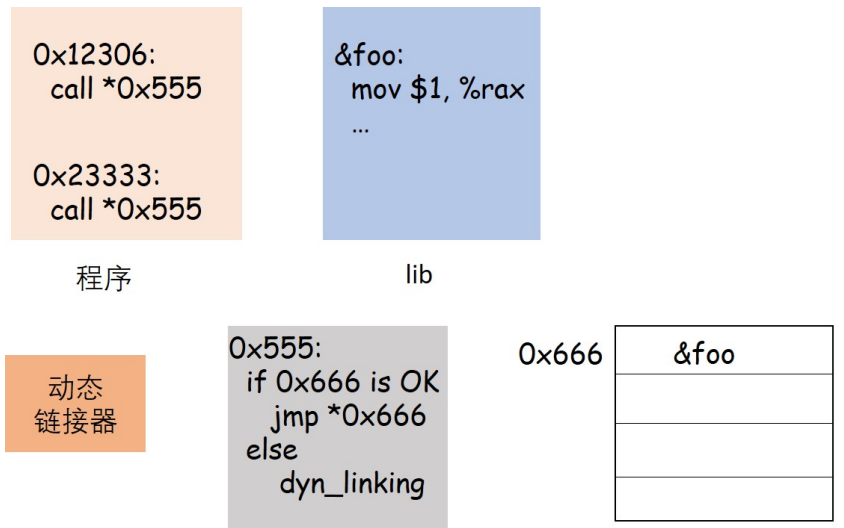
- 只有真正发生函数调用的时候才填表
- call foo -> call dyn_linking(foo) -> jmp foo

Lazy linking

事实上，函数的链接（填表）工作只需要进行一次

- call foo -> call dyn_linking(foo) -> jmp foo

缓存结果：增加一小段代码（stub），让它来判断我们是否已经完成链接



Almost done!

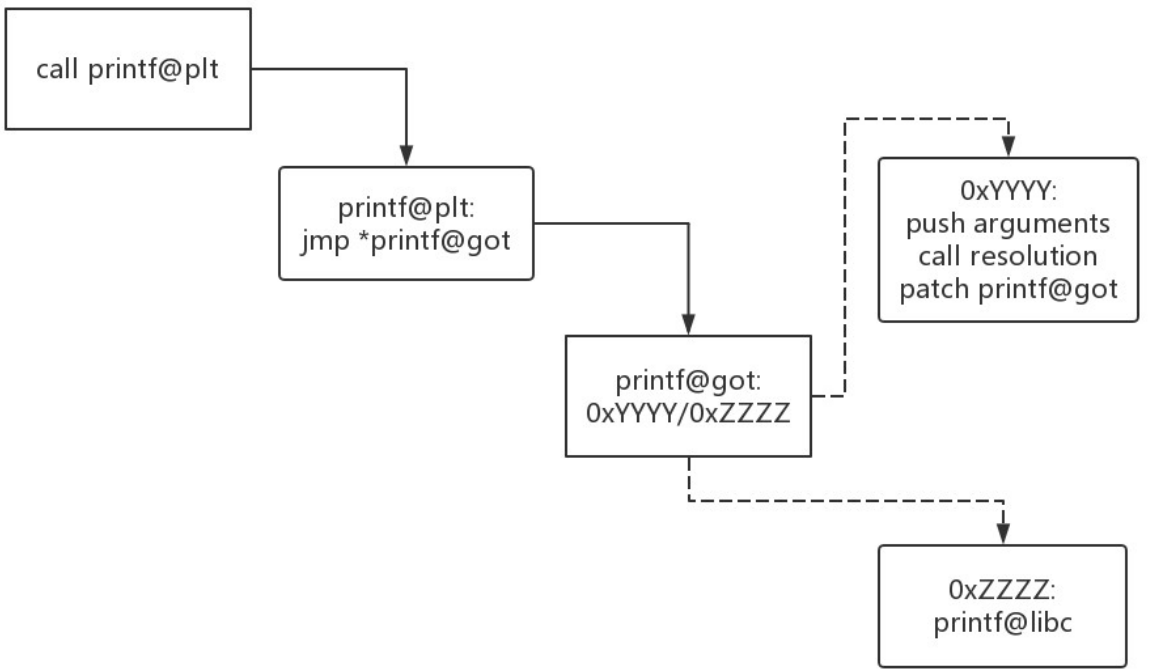
Polish 一下我们的表格

给我们的表格（Table）和那一小段代码（Stub）取个好听的名字

- GOT：Global Offset Table
- PLT： Procedure Linkage Table

Example

调用 printf 为例：



及时回顾

在我们设计动态链接的过程中，

- 动态 patch 函数调用地址（问题）
- 用表格记录信息，然后完成链接过程（解决方案）
- Add an indirection level （优化 -- 表格冗余）
- Lazy evaluation（优化 -- 解析冗余）
- Cache（优化 -- 调用链接器冗余）

动态链接 -- 数据

我们只说到代码，那么数据呢?

```
extern int bar;
bar += 1;
```

同样的，我们把需要动态链接的变量记录到 GOT 表中。

- mov `bar`, %rax

对于变量，我们就不采用 lazy linking 了

然而，实际我们用到的动态链接还有很多其他的细节 / 优化

- R_X86_64_COPY
- DYNAMIC SYMBOL TABLE
- ...

回到问题

在 PIE 编译下的程序，入口地址在哪里呢?

- 入口地址也被动态链接，只有运行时候知道
- 把整个程序所有的函数做作为可动态链接
- 很好理解啦

相比较：在没有 PIE 的程序编译下

- 只有动态链接库中的函数能够进行动态链接

优点：ASLR（Address Space Layout Randomization）更安全啦

动态链接器哪儿来的？

我们只是执行程序 `./a.out`，

- 那么这个动态链接器是如何介入其中，帮助我们的?

答案是：

- 操作系统在加载程序的时候顺便加载了链接器

我们下面会在 kernel source code 中找寻更精确的答案

理论部分结束

- ELF format 探索
- C++ 初始化代码
- 动态链接
- execve 代码阅读

谢谢！