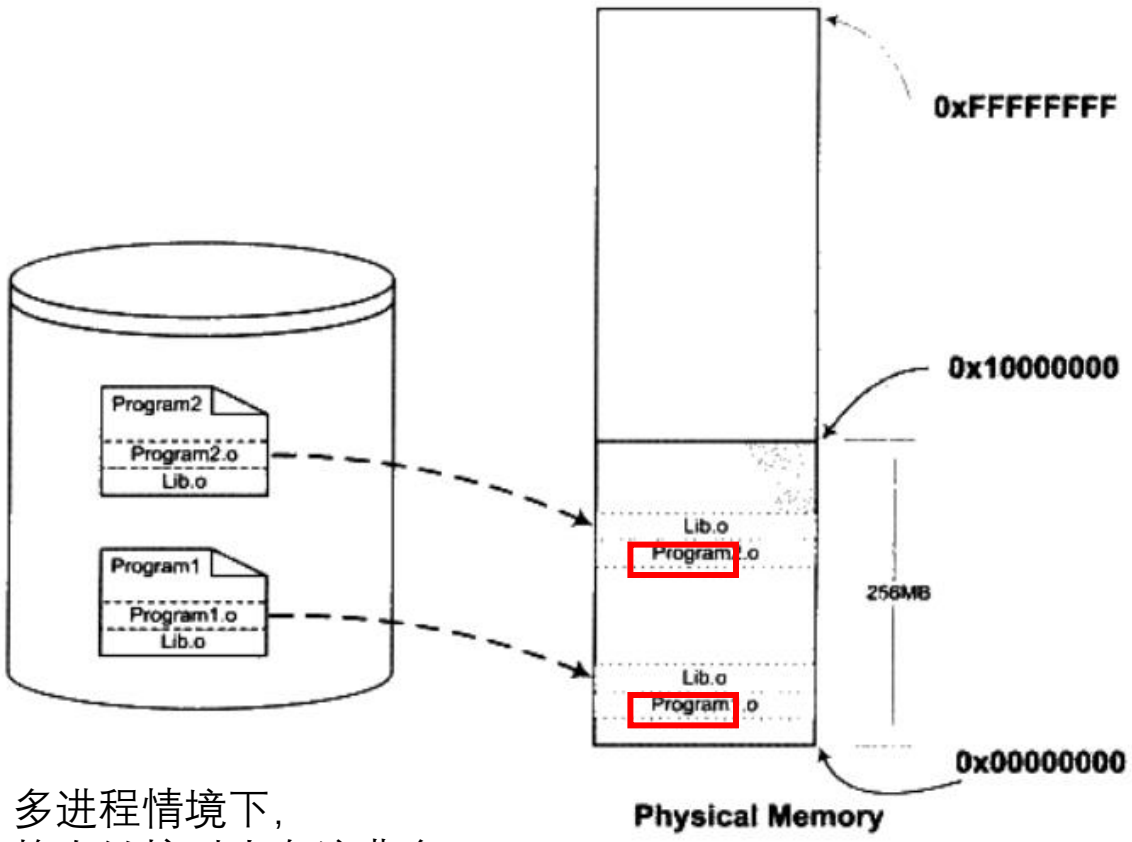
动态链接

目录

- 动态链接
- 延迟绑定
- 打桩

为什么需要动态链接?



多进程情境下，
静态链接对内存浪费多

静态链接：link time

重复拷贝公用库，浪费大量内存空间：

每个.o文件都需要链接所有其引用了的函数所属的库，产生大量不必要的重复，浪费宝贵的内存

更新程序不便：

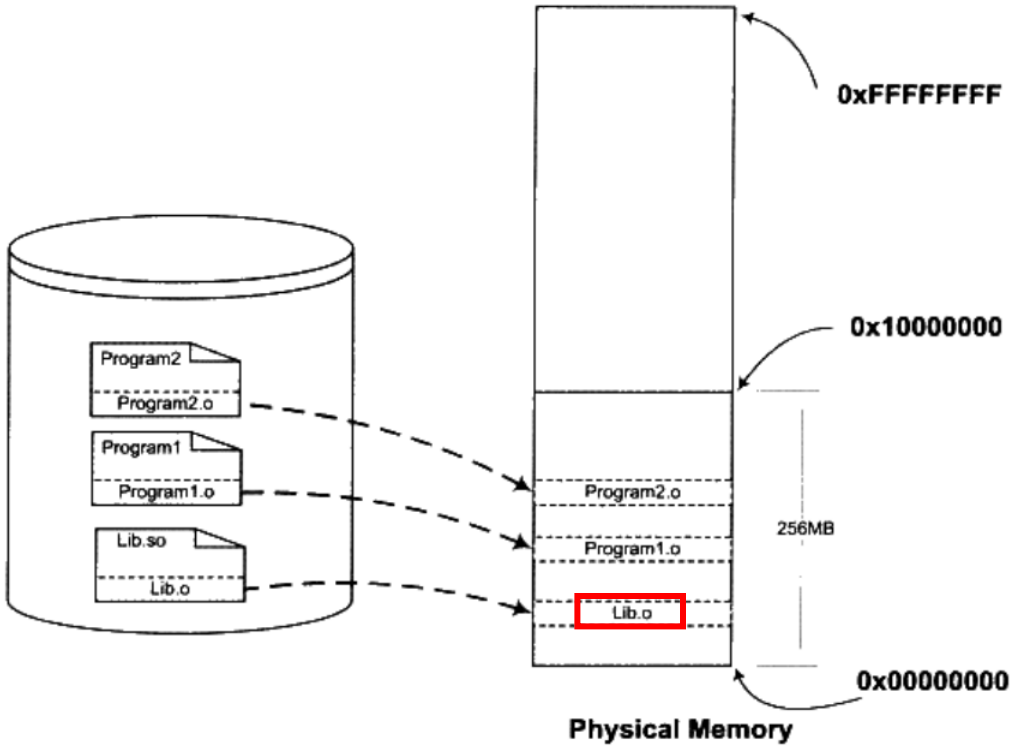
每次库更新时，.o文件发布者需要将文件与新的库重新链接后发布

动态链接：load time & runtime

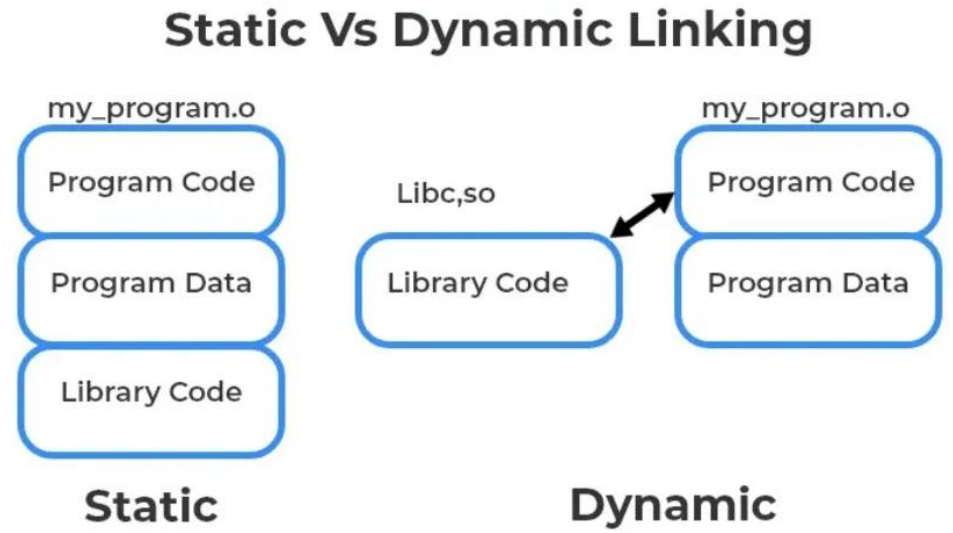
把程序的模块相互分割开，作为独立的文件进行使用，而非静态地链接在一起：不对组成程序的目标文件进行链接，等到程序要运行时才进行链接。

动态链接

- 动态链接（Dynamic Linking），把链接过程推迟到了运行时再进行，在可执行文件**装载时或运行时**由操作系统的装载程序加载库。



理想中动态链接应该是这样的：
程序和它们需要调用的库被独立地加载到内存中，
运行时程序按需在库中寻找到被调用的变量与函数



共享库(.so/.dll)文件

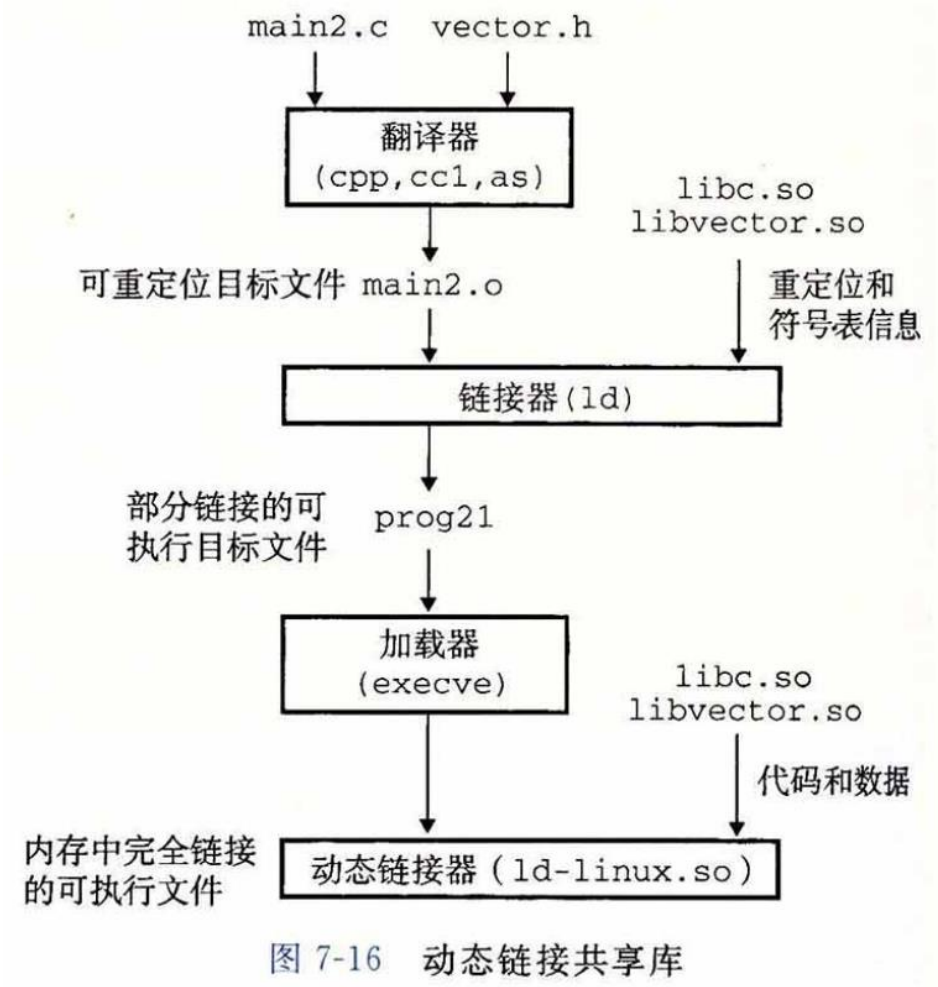


图 7-16 动态链接共享库

创建一个可执行文件时，进行一定的静态链接；在程序加载、运行时，完成动态链接的过程

在**可执行文件**prog21中，并没有来自libvector.so的代码与数据节，仅包含了来自动态共享库的重定位和符号表信息，用于在动态链接过程中寻址

-fpic 指示编译器生成**位置无关代码**
-shared 指示链接器创建一个**共享的目标文件**

```
linux> gcc -shared -fpic -o libvector.so addvec.c multvec.c
```

gcc -fpic -shared -o output.so **-L /path/to/libc.so /path/file1.o** file2.o

动态链接如何寻址?

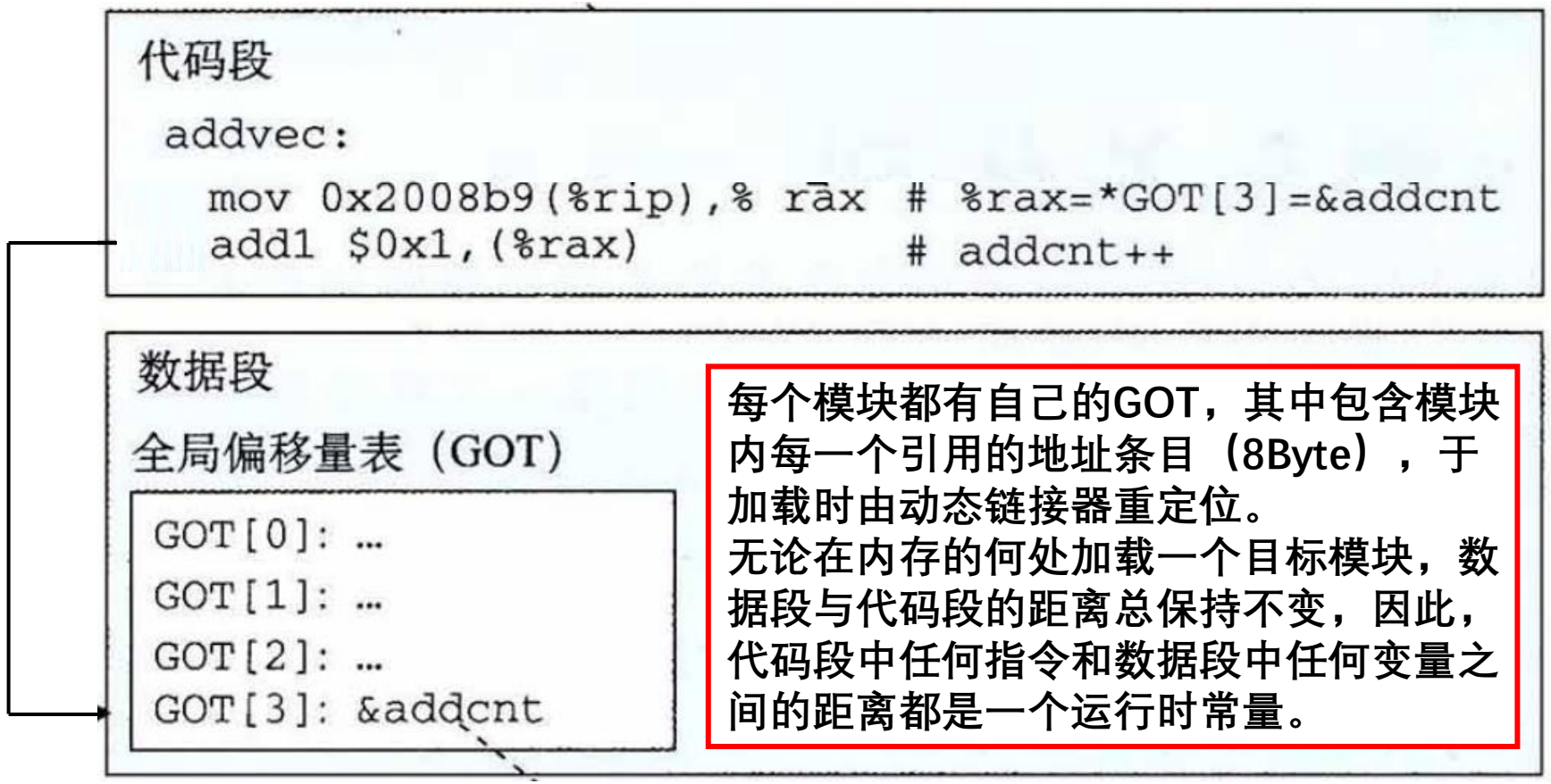
动态链接中，连接过程在加载后进行，然而我们已加载的代码段位于.rodata段(**不能对代码段进行修改**），因此静态链接时介绍的**符号重定位**的方法不能在加载后使用，这种方法需要依靠修改的代码段中的绝对地址信息完成。

动态链接寻址的核心思想是通过增加一个间接层，也就是**GOT**(Global Offset Table，全局偏移量表)和**PLT**(Procedure Linkage Table，过程链接表)，来辅助**动态链接器**完成对变量和函数的间接寻址。

利用GOT与PLT，避免使用绝对地址对库函数与变量进行寻址的代码称为**PIC**（Position Independent Code）、可以加载而无需重定位的代码，即**位置无关代码**。

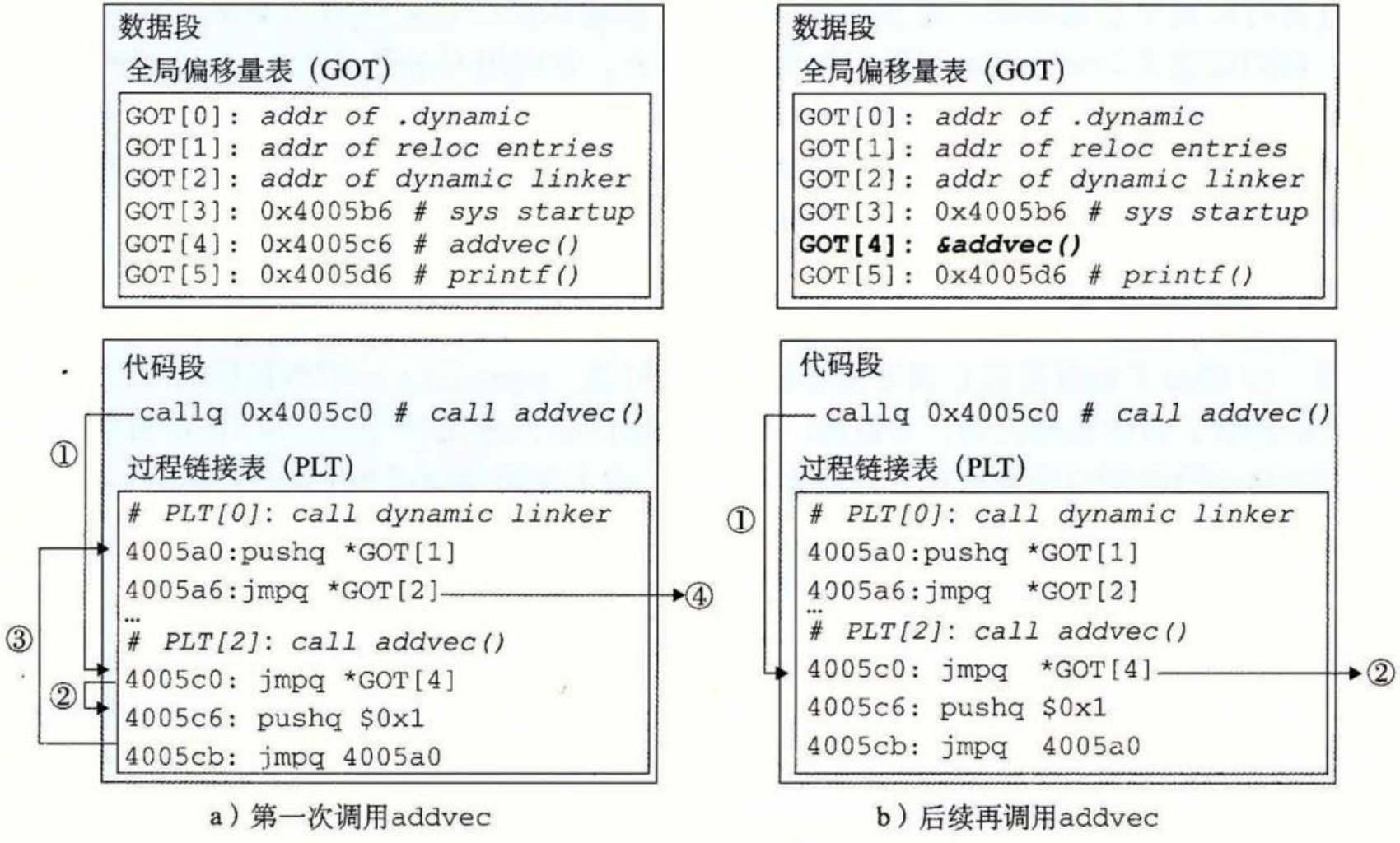
PIC寻址：PIC数据引用

运行时GOT[3]和addl指令之间的固定距离是0x2008b9



每个模块都有自己的GOT，其中包含模块内每一个引用的地址条目（8Byte），于加载时由动态链接器重定位。
无论在内存的何处加载一个目标模块，数据段与代码段的距离总保持不变，因此，代码段中任何指令和数据段中任何变量之间的距离都是一个运行时常量。

PIC寻址：PIC函数调用，延迟绑定



跳转表
模拟call指令

A. 以下程序可以使用 gcc dl.c -ldl 编译并正常运行。如果缺少了 -ldl 标志，链接时会报错 undefined reference to `dlopen'。基于你对于动态链接的理解，请分析出以下说法中<u>不正确</u>的一项。

```
// dl.c
#include <dlfcn.h>

const char *path = "/lib/libc.so";
int (*printf)(const char *x);

int main() {
  // 加载共享库
  void *handle = dlopen(path, RTLD_NOW);
  // 解析符号 "printf" 并返回地址
  printf = dlsym(handle, "printf");
  // 调用
  printf("2022 is coming!\n");
  // 关闭共享库
  dlclose(handle);
}
```

A. 该机器上 libdl.so 模块中包含符号名为 dlopen 的动态链接符号表条目
B. 在 a.out 文件中包含 printf 的 PLT 条目和相应的 GOT 条目
C. 在 a.out 文件中包含 dlopen 的 PLT 条目和相应的 GOT 条目
D. 如果使用 gcc -ldl dl.c 编译程序，则会在链接时发生同样错误

答案：B

-l选项的作用是指定链接时要使用的库文件，-ldl即在链接时使用libdl.so文件
不使用libdl.so时报错，说明dlopen函数定义来自于该共享库

A 正确，libdl.so中定义了dlopen函数
B 错误，printf在dl.c中是一个未初始化的全局变量（函数指针），而非函数，无PLT条目
C 正确，dlopen是dl.c中被引用的函数
D 正确，gcc按命令行顺序解析库

库打桩机制：编译时打桩

code/link/interpose/mymalloc.c

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

code/link/interpose/mymalloc.c

c) mymalloc.c 中的包装函数

设置编译选项

打桩函数内部不要打桩，即mymalloc.c中要使用原始的malloc函数，不然会造成循环调用

```
void* int_add_func(void* wParam)
{
    INT_PARAM* lParam = (INT_PARAM*)wParam;
    int res = lParam->param1 + lParam->param2;

    printf("result = %d\n", res);
}

void* double_add_func(void* wParam)
{
    DOUBLE_PARAM* lParam = (DOUBLE_PARAM*)wParam;
    double res = lParam->param1 + lParam->param2;

    printf("result = %f\n", res);

}

void* add_func(ADDFUNC f, void* wParam)
{
    return f(wParam);
}
```

假如 码中
的ma
mallo 程序
进行

-D 参
-I 选 算
库进行链接，实现调用我们修改过的打包函数
的功能

linu c.c
linu oc.o

```
add_func(int_add_func, &val1);

add_func(double_add_func, &val2);
```

链接时打桩&运行时打桩

- __wrap f标志进行**链接时打桩**：
  - 将对f的引用解析为__wrap_f将对__real_f的引用解析为f
  - 提前声明好__wrap_f与__real_f，如在main函数中调用malloc，将会去调用__wrap_malloc，而__real_malloc将会被解析成真正的malloc，从而达到“偷梁换柱”的效果。

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

-Wl, option 将option传递给链接器，***option,Filename*** 之间的逗号会被替换为空格

- **运行时打桩**：通过设置LD_PRELOAD，达到在加载一个动态库或者解析一个符号时，先从LD_PRELOAD指定的目录下的库去寻找需要的符号，然后再去其他库中寻找。

```
linux> LD_PRELOAD="./mymalloc.so" ./intr
```

运行时将**LD_PRELOAD**环境变量设置为**共享库路径名**，优先寻找

动态链接的问题：运行时依赖与版本冲突

运行时依赖：

动态链接意味着可执行文件在运行时需要依赖外部的共享库文件，这实际上
接而言）。

版本冲突：

当多个程序依赖
同的版本，进而
同步更新，否则
（DLL Hell，本

名称	修改日期	类型	大小
settings-layout.json	2023/1/12 9:57	JSON 文件	
SmokeAPI.cache.json	2023/9/1 20:00	JSON 文件	
cream_api.ini	2023/11/6 19:22	配置设置	
.dist.v1.json	2023/11/11 3:00	JSON 文件	
d3d9.dll	2023/11/11 3:00	应用程序扩展	
README.md	2023/11/11 3:00	Markdown 源文件	
version.dll	2023/11/11 3:00	应用程序扩展	
eu4.exe	2023/11/15 23:17	应用程序	
eu4_profiling.exe	2023/11/15 23:17	应用程序	
launcher-settings.json	2023/11/15 23:17	JSON 文件	