- 第 3 步。在把 addvec 的 ID(0x1)压入栈中之后，PLT[2]跳转到 PLT[0]。
- 第 4 步。PLT[0]通过 GOT[1]间接地把动态链接器的一个参数压入栈中，然后通过 GOT[2]间接跳转进动态链接器中。动态链接器使用两个栈条目来确定 addvec 的运行时位置，用这个地址重写 GOT[4]，再把控制传递给 addvec。

图 7-19b 给出的是后续再调用 addvec 时的控制流：

- 第 1 步。和前面一样，控制传递到 PLT[2]。
- 第 2 步。不过这次通过 GOT[4]的间接跳转会将控制直接转移到 addvec。

7.13　库打桩机制

Linux 链接器支持一个很强大的技术，称为*库打桩*(library interpositioning)，它允许你截获对共享库函数的调用，取而代之执行自己的代码。使用打桩机制，你可以追踪对某个特殊库函数的调用次数，验证和追踪它的输入和输出值，或者甚至把它替换成一个完全不同的实现。

下面是它的基本思想：给定一个需要打桩的*目标函数*，创建一个*包装函数*，它的原型与目标函数完全一样。使用某种特殊的打桩机制，你就可以欺骗系统调用包装函数而不是目标函数了。包装函数通常会执行它自己的逻辑，然后调用目标函数，再将目标函数的返回值传递给调用者。

打桩可以发生在编译时、链接时或当程序被加载和执行的运行时。要研究这些不同的机制，我们以图 7-20a 中的示例程序作为运行例子。它调用 C 标准库(libc.so)中的 malloc 和 free 函数。对 malloc 的调用从堆中分配一个 32 字节的块，并返回指向该块的指针。对 free 的调用把块还回到堆，供后续的 malloc 调用使用。我们的目标是用打桩来追踪程序运行时对 malloc 和 free 的调用。

7.13.1　编译时打桩

图 7-20 展示了如何使用 C 预处理器在编译时打桩。mymalloc.c 中的包装函数(图 7-20c)调用目标函数，打印追踪记录，并返回。本地的 malloc.h 头文件(图 7-20b)指示预处理器用对相应包装函数的调用替换掉对目标函数的调用。像下面这样编译和链接这个程序：

```
linux> gcc -DCOMPILETIME -c mymalloc.c
linux> gcc -I. -o intc int.c mymalloc.o
```

由于有-I.参数，所以会进行打桩，它告诉 C 预处理器在搜索通常的系统目录之前，先在当前目录中查找 malloc.h。注意，mymalloc.c 中的包装函数是使用标准 malloc.h 头文件编译的。

运行这个程序会得到如下的追踪信息：

```
linux> ./intc
malloc(32)=0x9ee010
free(0x9ee010)
```

7.13.2　链接时打桩

Linux 静态链接器支持用--wrap f 标志进行链接时打桩。这个标志告诉链接器，把对符号 f 的引用解析成__wrap_f(前缀是两个下划线)，还要把对符号__real_f(前缀是两个下划线)的引用解析为 f。图 7-21 给出我们示例程序的包装函数。