


这些声明表明,在叫做“.rodata”(只读数据,Read-Only Data)的目标代码文件的段中,应该有一组7个“四”字(8个字节),每个字的值都是与指定的汇编代码标号(例如.L3)相关联的指令地址。标号.L4标记出这个分配地址的起始。与这个标号相对应的地址会作为间接跳转(第5行)的基地址。

不同的代码块(C标号 loc_A 到 loc_D 和 loc_def)实现了 switch 语句的不同分支。它们中的大多数只是简单地计算了 val 的值,然后跳转到函数的结尾。类似地,汇编代码块计算了寄存器%rdi 的值,并且跳转到函数结尾处由标号.L2 指示的位置。只有情况标号 102 的代码不是这种模式的,正好说明在原始 C 代码中情况 102 会落到情况 103 中。具体处理如下:以标号.L5 起始的汇编代码块中,在块结尾处没有 jmp 指令,这样代码就会继续执行下一个块。类似地,C 版本 switch_eg_impl 中以标号 loc_B 起始的块的结尾处也没有 goto 语句。

检查所有这些代码需要很仔细的研究,但是关键是领会使用跳转表是一种非常有效的实现多重分支的方法。在我们的例子中,程序可以只用一次跳转表引用就分支到 5 个不同的位置。甚至当 switch 语句有上百种情况的时候,也可以只用一次跳转表访问去处理。

 **练习题 3.30** 下面的 C 函数省略了 switch 语句的主体。在 C 代码中,情况标号是不连续的,而有些情况有多个标号。

```
void switch2(long x, long *dest) {
    long val = 0;
    switch (x) {
        :   Body of switch statement omitted
    }
    *dest = val;
}
```

在编译该函数时,GCC 为程序的初始部分生成了以下汇编代码,变量 x 在寄存器%rdi 中:

```
void switch2(long x, long *dest)
x in %rdi
1  switch2:
2      addq    $1, %rdi
3      cmpq    $8, %rdi
4      ja      .L2
5      jmp     *.L4(,%rdi,8)
```

为跳转表生成以下代码:

```
1  .L4:
2      .quad   .L9
3      .quad   .L5
4      .quad   .L6
5      .quad   .L7
6      .quad   .L2
7      .quad   .L7
8      .quad   .L8
9      .quad   .L2
10     .quad   .L5
```

根据上述信息回答下列问题:

- A. switch 语句内情况标号的值分别是多少?
- B. C 代码中哪些情况有多个标号?